Съдържание

1 Увод

Основна задача на лексическият анализатор е да чете символите на входния текст, да ги групира под формата на лексеми (тоукъни) и извежда като изход редица от тези лексеми. Лексемата е структура от данни, която съдържа нейния тип, текст и позиция във входния текст. Тази информация в последствие се подава на парсър, който от своя страна извършва синтактичният анализ на текста.

Лексическите анализатори могат да се използват и за други цели освен идентификация на лексемите, като на пример за премахване на сегменти от текст (като нови редове, интервали и пр.), броене на символи, или думи, както и за проверка за грешки.

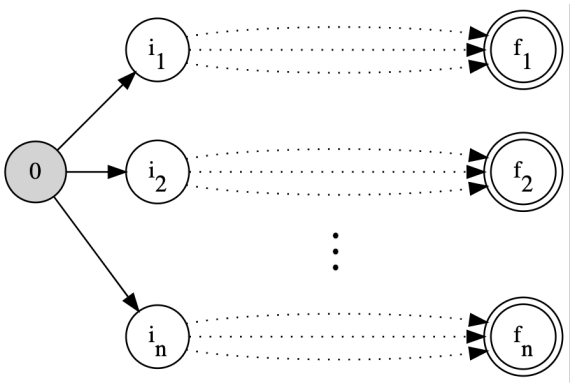
Лексемите се дефинират чрез регулярни изрази. Генератор за лексически анализ получава редица от регулярни изрази като вход и въз основа на тях строи лексически анализатор.

В тази работа представяме алгоритъм за конструкция на лексически анализатор по зададено множество от регулярни изрази. Анализаторът извлича лексемите за линейно време спрямо дължината на входния текст, като го сканира едновременно от ляво на дясно и от дясно на ляво използвайки бимашина.

1.1 Мотивация

Съществуващите генератори на лексически анализатори като Lex, Flex и ANTLR намират широко приложение в индустрията. Те позволяват на потребителя да подаде спецификация на лексемите (лексическа граматика) под формата на регулярни изрази и генерират програма, която извършва токенизацията входен текст спрямо тази спецификация.

Тези инструменти работят на сходен принцип. По регулярните изрази на всяко правило от граматиката, те строят крайни автомати, които в последствие се обединяват (Фигура 1.1). Токенизацията се извършва, като полученият автомат се симулира чрез сканиране на текста от ляво на дясно. Ако при прочетен символ, автоматът не може да направи преход към нито едно състояние, то последното посетено финално състояние определя лексемата, която да се изведе, автоматът преминава обратно в началното си състояние и сканирането на входния текст продължава от от символа, който е бил прочетен, когато автоматът се е намирал във въпросното финално състояние. Ако автоматът не може да продължи и междувременно не е посетено финално състояние, то входния текст не е коректен спрямо лексическата граматика.



Фигура 1.1: Краен автомат, получен от обединението на автоматите от лексическа граматика с n на брой правила.

Пример 1.1. Нека разгледаме следната спецификация:

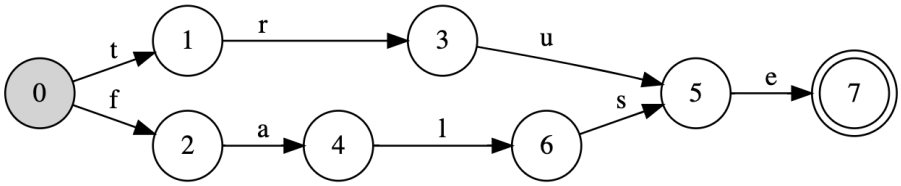
Token	Description
Id	`[a-zA-Z_][a-zA-Z0-9_]*`
Number	`[0-9]+(\.[0-9]+)?`
Boolean	`true|false`
Operator	`=|==|!=|<|<=|>|>=`
If	`if`
Else	`else`
Return	`return`
BraceOpen	`{`
BraceClose	`}`
WS	`[ \t\r\n]+`

Фигура 1.2: Лексическа граматика

Входният текст *"num_1=90.4"*, се разбива на следните лексеми:
(*num_1*, **Id**), (=, **Operator**), (*90.4*, **Number**).
Друг пример е думата *"if valid==true return 0"*, за която получаваме:
(*if*, **If**), (*' '*, **WS**), (*valid*, **Id**), (==, **Operator**), (*true*, **Boolean**), (*' '*, **WS**), (*return*, **Return**), (*' '*, **WS**), (*0*, **Number**).

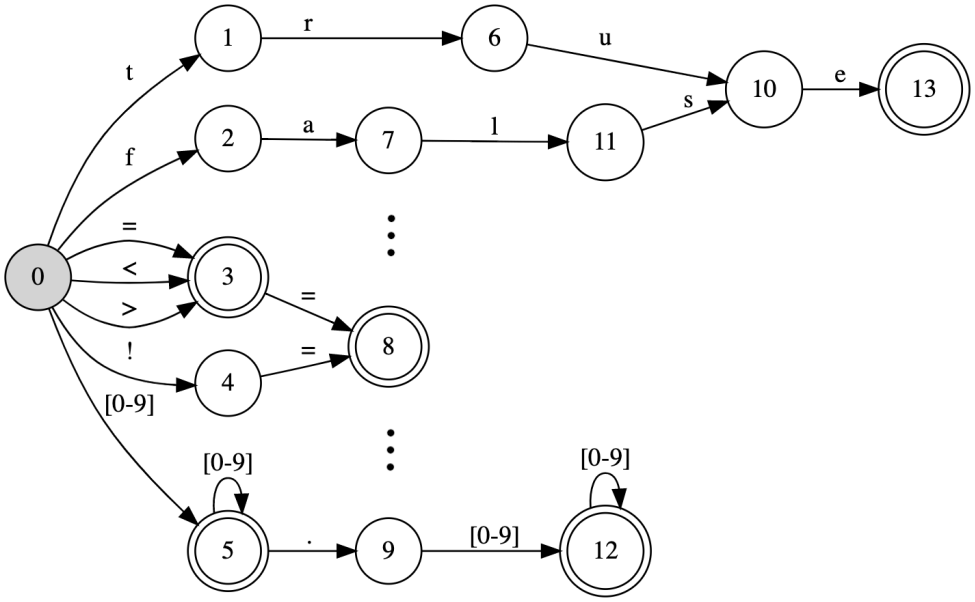
Всяко правило се представя чрез краен автомат, като на пример този за "Boolean"е изобразен на Фигура 1.3.

След като автоматите за всяко правило са построени, следващата стъпка е



Фигура 1.3: Краен автомат разпознаващ думите *true* или *false*

да се обединят в единствен краен автомат (Фигура 1.4), който се симулира по време на сканирането на входния текст.



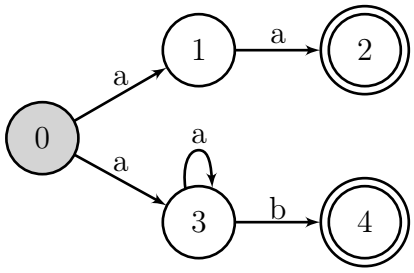
Фигура 1.4: Краен автомат, получен от обединението на автоматите от лексическата граматика. За яснота са изобразени само Boolean, Operator и Number.

Нека разгледаме входния текст *"1 > 0.99 == true"*. Преди да започнем да го четем, авоматът се намира в началното си състояние 0. Четем *'1'* и преминаваме в състояние 5. Следващият символ е *'>'*, за който няма преход. Намираме се във финално състояние на автомата на лексемата Number, съответно извеждаме (*1*, **Number**) и се връщаме в началното състояние. Аналогично от 0 имаме преход с *'>'*, който води до състояние 3, което е финално. Няма преход от 3 на *'0'*, съответно извеждаме (*>*, **Operator**) и се връщаме в 0. Прочитайки *'0'*, автоматът преминава в състояние 5, което е финално, но следващият символ е

'.', така че можем да продължим със симулацията като стигаме до състояние 12 и извеждаме (*'0.99'*, **Number**). Аналогично по пътя 0,3,8 ще изведем (*'=='*, **Operator**) и в последствие по 0,1,6,10,13 ще изведем (*'true'*, **Boolean**), като с това сме изчерпали символите в тескта и процедурата приключва.

В определени случаи този подход може да се окаже неефективен. Нека разгледаме следния прост пример.

Token	Description
A	`aa`
B	`a+b`



Фигура 1.5: Лексическа граматика и построеният по нея краен автомат.

Пример 1.2. Граматиката на Фигура 1.1 съдържа две правила. Правило **A** представлява единствено думата *"aa"*. Правило **B** обхваща думите, съдържащи *'a'* поне веднъж, които завършват с *'b'* като на пример *"ab "aab "aaab"* и т.н. Входният текст *"aabaaaa* се разбива на следните лексеми: (*'aab'*, **B**), (*'aa'*, **A**), (*'aa'*, **A**).

Нека разгледаме случая, в който сканираме текста *"aaaaaaaaaa"*, състоящ се от пет еднакви тоукъна - (*'aa'*, **A**). Очевидно, за да изведем **A** е необходимо да сканираме текста до самия му край, за да се уверим, че не съществува *'b'*. Това се случва за всеки изведен тоукън, което води до неоптимална сложност на процедурата от $\mathcal{O}(n^2)$.

Лексическият анализ чрез бимашина се справя с този проблем като сканирането на текста се случва едновременно от ляво на дясно и от дясно на ляво, с което се гарантира линейно време на изпълнение. Целта на тази работа е да се представи конструкция на такава бимашина по зададена лексическа граматика и алгоритъм за осъществяване на лексическия анализ.

2 Основни дефиниции

2.1 Крайни автомати

Дефиниция 2.1. *Краен автомат* дефинираме като петорка $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$, където

- Σ е *крайна азбука от символи*
- Q е *крайно множество от състояния*
- $I \subseteq Q$ е *множество от начални състояния*
- $F \subseteq Q$ е *множество от финални състояния*
- $\Delta \subseteq Q \times \Sigma \times Q$ е *релация на прехода*

Тройки от вида $\langle q_1, m, q_2 \rangle \in \Delta$ наричаме *преходи* и казваме, че започва състояние q_1, има етикет m и завършва в състояние q_2. Алтернативно, тези преходи обозначаваме като $q_1 \rightarrow^m q_2$.

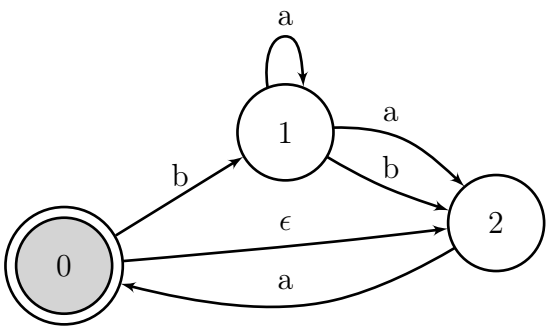
Дефиниция 2.2. Нека $\mathcal{A}$ е краен автомат. *Разширена релация на прехода* $\Delta^* \subseteq Q \times \Sigma^* \times Q$ дефинираме индуктивно:

- $\langle q, \epsilon, q \rangle \in \Delta^*$ за всяко $q \in Q$
- $\langle q_1, wa, q_2 \rangle \in \Delta^*$ за всяко $q_1, q_2, q \in Q$, $a \in \Sigma, w \in \Sigma^*$, ако $\langle q_1, w, q \rangle \in \Delta^*$ и $\langle q, a, q_2 \rangle \in \Delta$

Дефиниция 2.3. Нека $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ е краен автомат. *Път* в $\mathcal{A}$ наричаме крайна редица от преходи с дължина $k > 0$

$$\pi = q_0 \rightarrow^{a_1} q_1 \rightarrow^{a_2} \ldots \rightarrow^{a_k} q_k$$

където $\langle q_{i-1}, a_i, q_i \rangle \in \Delta$ за $i = 1 \ldots k$. Казваме, че *пътят* започва от състояние q_0 и завършва в състояние q_k. Елементите $q_0, q_1, \ldots, q_k$ наричаме *състояния на пътя*, а думата $w = a_1 a_2 \ldots a_k$ наричаме *етикет на пътя*.
Успешен път в автомата е *път*, който започва от начално състояние и завършва във финално състояние.



Фигура 2.1: Недетерминиран краен автомат

Пример 2.1. Нека е зададена азбука $\Sigma = \{a, b, \epsilon\}$ и автомат $\mathcal{A}$ над Σ със състояния $Q = \{0, 1, 2\}$, начални $I = \{0\}$, финални $F = \{0\}$ и релация на прехода

$$\Delta = \{\langle 0, b, 1\rangle, \langle 0, \epsilon, 2\rangle, \langle 1, a, 1\rangle, \langle 1, a, 2\rangle \langle 1, b, 2\rangle, \langle 2, a, 0\rangle\}$$

$\mathcal{A}$ е изобразен на Фигура 2.1. $0 \to^b 1 \to^a 1 \to^b 2 \to^a 0$ е успешен път, разпознавайки думата *baba*.

Дефиниция 2.4. Нека $\mathcal{A}$ е краен автомат. Множеството от етикети на всички успещни пътища в $\mathcal{A}$ наричаме *език на* $\mathcal{A}$ и обозначаваме като $L(\mathcal{A})$.

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \exists i \in I, f \in F : \langle i, w, f\rangle \in \Delta^*\}$$

Дефиниция 2.5. Нека $\mathcal{A}_1$ и $\mathcal{A}_2$ са крайни автомати. Казваме, че $\mathcal{A}_1$ е еквивалентен на $\mathcal{A}_2$ ($\mathcal{A}_1 \equiv \mathcal{A}_2$), ако езиците им съвпадат ($L(\mathcal{A}_1) = L(\mathcal{A}_2)$)

Дефиниция 2.6. Нека $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta\rangle$ е краен автомат. Автоматът $\mathcal{A}' = \langle Q, F, I, \Delta'$, където $\Delta' = \{\langle q, a, p\rangle \mid \langle p, a, q\rangle \in \Delta\}$ наричаме *огледален* на $\mathcal{A}$. За всяка дума $w = w_1 w_2 \dots w_k \in L(\mathcal{A})$ е в сила $w' = w_k \dots w_2 w_1 \in L(\mathcal{A}')$.

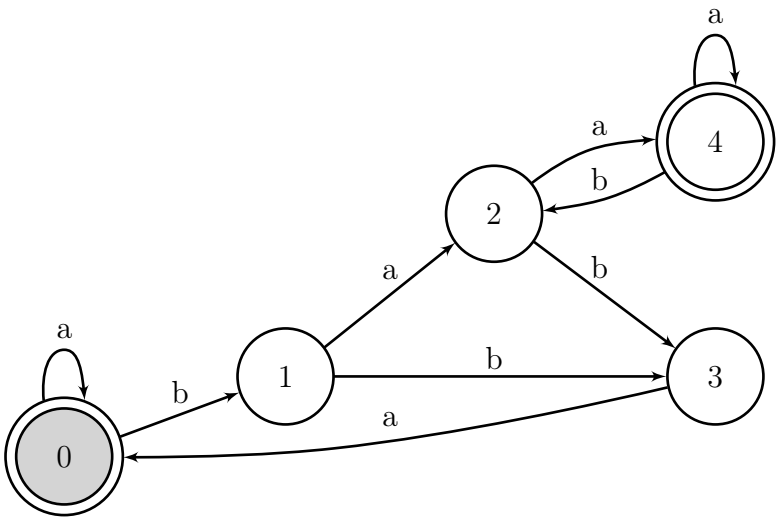
Дефиниция 2.7. Краен автомат $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta\rangle$ е *детерминиран*, ако:

- $\mathcal{A}$ има единствено начално състояние $I = \{q_0\}$.
- За всяко $q_1 \in Q$ и символ $a \in \Sigma$, съществува не повече от едно $q_2 \in Q$, такова че $\langle q_1, a, q_2\rangle \in \Delta$.

Иначе казано, релацията на прехода може да се представи като частична функция $\delta : Q \times \Sigma \to Q$ и *детерминираните автомати* можем преставим в следния вид

$$\mathcal{A}_D = \langle \Sigma, Q, q_0, F, \delta\rangle$$

Предимството на *детерминираните автомати* се изразява в това, че могат да разпознават дали дума w принадлежи на езика на автомата $L(\mathcal{A}_D)$ за линейно време спрямо дължината ѝ - $O(|w|)$, но в определени случаи могат да имат експоненциален брой състояния спрямо еквивалентният им недетерминиран автомат.



Фигура 2.2: Детерминиран краен автомат

Дефиниция 2.8. Нека $\mathcal{A}_D = \langle \Sigma, Q, q_0, F, \delta \rangle$ е *детерминиран краен автомат. Разширена функция на прехода* $\delta^* : Q \times \Sigma^* \to Q$ дефинираме индуктивно:

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$, където $a \in \Sigma, w \in \Sigma^*$

Теорема 2.1. За всеки краен автомат $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$, съществува еквивалентен на него, детерминиран краен автомат $\mathcal{A}_D$, където $L(\mathcal{A}) = L(\mathcal{A}_D)$.

Доказателство. Нека $\mathcal{A}$ е краен автомат, на който сме премахнали ϵ-преходите. Строим *детерминиран краен автомат* $\mathcal{A}_D = \langle \Sigma, 2^Q, I, F_D, \delta \rangle$, където:

- $F_D = \{S \in 2^Q \mid S \cap F \neq \emptyset\}$
- $\delta(S, a) = \{q \in Q \mid \exists p \in S : \langle p, a, q \rangle \in \Delta\}$

С индукция по дължината на w, ще покажем, че за произволна дума $w \in \Sigma^*$ твръденията $\exists i \in I : \langle i, w, p\rangle \in \Delta^*$ и $p \in \delta^*(I, w)$ са еквивалентни:

- *База*: за $|w| = 0$ имаме $w = \epsilon$. Тогава $\exists i \in I : \langle i, \epsilon, i\rangle \in \Delta^*$. Тъй като $\mathcal{A}$ няма ϵ-преходи, то $\delta^*(I, \epsilon) = I$.

- *Индукция*: $w = w'a, a \in \Sigma, |w| = |w'| + 1$.
  ($\Rightarrow$) Нека допуснем, че $\exists i \in I$, така че $\langle i, w'a, p\rangle \in \Delta^*$, което значи, че $\exists p' \in Q : \langle p', a, p\rangle \in \Delta$. По индуктивното предположение знаем, че и $p' \in \delta^*(I, w')$ и от дефиницията на δ следва, че $p \in \delta(\delta^*(I, w'), a)$.
  ($\Leftarrow$) Нека допуснем, че $p \in \delta^*(I, w'a)$. Тогава $\exists p' : p' \in \delta^*(I, w')$. От индуктиното предположение знаем, че $\langle i, w', p'\rangle \in \Delta^*$ и от дефинициите на δ и Δ следва, че $\langle p', a, p\rangle \in \Delta$, от където следва, че $\exists i \in I : \langle i, w'a, p\rangle \in \Delta^*$.

Така можем да заключим, че за всяка дума $w \in L(\mathcal{A})$, $\exists i \in I, \exists f \in F : \langle i, w, f\rangle \in \Delta^*$ е изпълнено, че $\delta^*(I, w) \in F_D$, следователно $w \in L(\mathcal{A}_D)$ и $L(\mathcal{A}) = L(\mathcal{A}_D)$. $\square$

2.2 Крайни преобразуватели

Дефиниция 2.9. *Краен преобразувател* дефинираме като петорка $\mathcal{T} = \langle \Sigma_1^* \times \Sigma_2^*, Q, I, F, \Delta\rangle$, където

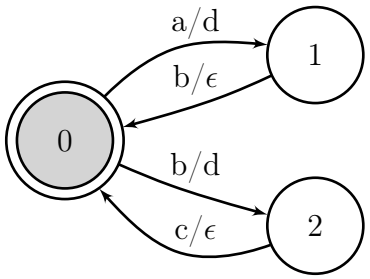
- Σ_1, Σ_2 са *крайни азбуки от символи*
- Q е *крайно множество от състояния*
- $I \subseteq Q$ е *множество от начални състояния*
- $F \subseteq Q$ е *множество от финални състояния*
- $\Delta \subseteq Q \times (\Sigma_1^* \times \Sigma_2^*) \times Q$ е *релация на прехода*

Тройки от вида $\langle q_1, \langle w, m\rangle, q_2\rangle \in \Delta$ наричаме *преходи* и казваме, че започва състояние q_1, има етикет по горната лента w и по долната лента m и завършва в състояние q_2. Алтернативно, тези преходи обозначаваме като $q_1 \rightarrow_m^w q_2$.

Дефиниция 2.10. Нека $\mathcal{T} = \langle \Sigma_1^* \times \Sigma_2^*, Q, I, F, \Delta\rangle$ е краен преобразувател. *Път* в $\mathcal{T}$ наричаме крайна редица от преходи с дължина $k > 0$

$$\pi = q_0 \rightarrow_{m_1}^{w_1} q_1 \rightarrow_{m_2}^{w_2} \ldots \rightarrow_{m_k}^{w_k} q_k$$

където $\langle q_{i-1}, \langle w_i, m_i\rangle, q_i\rangle \in \Delta$ за $i = 1 \dots k$. Казваме, че *пътят* започва от състояние q_0 и завършва в състояние q_k. Елементите $q_0, q_1, \dots, q_k$ наричаме *състояния на пътя*, а думата $w = w_1 w_2 \dots w_k$ наричаме *входна дума* на пътя, а $m = m_1 m_2 \dots m_k$ е *изходна дума* на пътя.
Успешен път в преобразувателя започва от начално състояние и завършва във финално състояние.



Фигура 2.3: Краен преобразувател

Пример 2.2. Нека са зададени азбукаи $\Sigma_1 = \{a, b, c\}, \Sigma_2 = \{d, \epsilon\}$ и преобразувател $\mathcal{T}$ над $\Sigma_1^* \times \Sigma_2^*$ със състояния $Q = \{0, 1, 2\}$, начални $I = \{0\}$, финални $F = \{0\}$ и релация на прехода $\Delta = \{\langle 0, \langle a, d\rangle, 1\rangle, \langle 1, \langle b, \epsilon\rangle, 0\rangle, \langle 0, \langle b, d\rangle, 2\rangle, \langle 2, \langle c, \epsilon\rangle, 0\rangle\}$.
$\mathcal{T}$ е изобразен на Фигура 2.3.
$0 \to_d^b 2 \to_\epsilon^c 0 \to_d^a 1 \to_\epsilon^b 0$ е успешен път, превеждайки думата *bcab* в *dd*.

Дефиниция 2.11. Нека $\mathcal{T} = \langle \Sigma_1^* \times \Sigma_2^*, Q, I, F, \Delta\rangle$ е краен преобразувател. *Подлежащ автомат* на $\mathcal{T}$ дефинираме като $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta'\rangle$, където $\Delta' = \{\langle p, w, q\rangle \mid \langle p, \langle w, m\rangle, q\rangle \in \Delta\}$

2.3 Регулярни езици и релации

Дефиниция 2.12. *Регулярен език* е множество над крайна азбука Σ, което дефинираме индуктивно:

- $\emptyset$ е регулярен език.
- ако $a \in \Sigma$, то $\{a\}$ е регулярен език.
- ако $L_1, L_2 \subseteq \Sigma^*$ са регулярни езици, то
  - $L_1 \cup L_2$ (обединение)

  - $L_1 \cdot L_2 = \{a \cdot b \mid a \in L_1, b \in L_2\}$ (конкатенация)
  - $L_1^* = \bigcup_{i=0}^{\infty} L^i$ (звезда на Клини)

  са също регулярни езици.

- Не съществуват други регулярни езици.

Регулярните езици са също така *затворени* относно операциите *допълнение* ($\Sigma^* \setminus L_1$), *разлика* ($L_1 \setminus L_2$), *обръщане* (L_1^{-1}) и *сечение* ($L_1 \cap L_2$).

Дефиниция 2.13. *Двоична регулярна стрингова релация* дефинираме индуктивно като множество от двойки над крайни азбуки Σ_1, Σ_2:

- $\emptyset$ е регулярна релация.
- Ако $a \in \Sigma_1 \times \Sigma_2$, то $\{a\}$ е регулярна релация.
- Ако R_1, R_2 са регулярни релации, то:
  - $R_1 \cup R_2$ (обединение)
  - $R_1 \cdot R_2 = \{a \cdot b \mid a \in R_1, b \in R_2\}$ (конкатенация)
  - $R_1^* = \bigcup_{i=0}^{\infty} R^i$ (звезда на Клини)

  са също регулярни релации.

- Не съществуват други регулярни релации.

Дефиниция 2.14. Нека $R_1, R_2, \in \Sigma^* \times \Sigma^*$ са *двоични стрингови релации. Конкатенацията* на R_1 и R_2 дефинираме както следва

$$R_1 \cdot R_2 = \{\langle u_1 \cdot v_1, u_2 \cdot v_2 \rangle \mid \langle u_1, u_2 \rangle \in R_1, \langle v_1, v_2 \rangle \in R_2\}$$

Дефиниция 2.15. *Регулярен израз* наричаме дума над крайна азбука $\Sigma \cup \{(,), |, *\}$

- ϵ е регулярен израз.
- Ако $a \in \Sigma$, то a е регулярен израз.
- Ако E_1, E_2 са регулярни изрази, то $E_1|E_2$ и E_1E_2, E_1^* също са регулярни изрази.
- Не съществуват други регулярни изрази.

Всеки регулярен израз има съответстващ регулярен език:

- $L(\epsilon) = \emptyset$
- $L(a) = \{a\}, a \in \Sigma$
- $L(E_1|E_2) = L(E_1) \cup L(E_2)$
- $L(E_1E_2) = L(E_1) \cdot L(E_2)$
- $L(E_1^*) = L(E_1)^*$

Пример 2.3. `(a|b)*c` е *регулярен израз*, разпознаващ думите *c, ac, bc, aac, abc, abbc, bababbc ...*

Теорема 2.2. *(Клини)* За всеки регулярен израз E съществува краен автомат $\mathcal{A}$, за който $L(E) = L(\mathcal{A})$.

Теорема 2.3. Всяка двоична регулярна стрингова релация може да се представи, чрез класически краен преобразувател.

Пример 2.4. $R = \{\langle ab, d\rangle, \langle bc, d\rangle\}^*$ е *регулярна релация*, която е представена чрез чрез крайния преобразувател $\mathcal{T}$ на Фигура 2.3.
С *dom(R)* бележим домейна на R, който изразяваме чрез *подлежащия автомат* на $\mathcal{T}$, $dom(R) = \{ab, bc, abab, abbc, bcab \dots\}$. С *range(R)* обозначаваме кодомейна на релацията $range(R) = \{\epsilon, d, dd, ddd, dddd \dots\}$.

Пример 2.5. Нека L е *регулярен език*. $Id(L) = \{\langle w, w\rangle \mid w \in L\}$ е *регулярна релация.*

Дефиниция 2.16. *Регулярна стрнгова функция* наричаме регулярна стрингова релация, която е частична функция.

2.4 Бимашини

Дефиниция 2.17. *Класическа бимашина* дефинираме като тройка $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$, където

- $\mathcal{A}_L = \langle \Sigma, Q_L, s_L, Q_L, \delta_L\rangle$ и $\mathcal{A}_R = \langle \Sigma, Q_R, s_R, Q_R, \delta_R\rangle$ са *детерминирани крайни автомати* и ги наричаме съответно *ляв и десен автомат* на бимашината. Всички състояния на тези автомати са финални.

- $\psi : (Q_L \times \Sigma \times Q_R) \to \Sigma^*$ е частична функция, която наричаме *изходна функция.*

Дефиниция 2.18. Нека $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ е класическа бимашина, Σ е азбуката на на автоматите $\mathcal{A}_L$ и $\mathcal{A}_R$ и $w = a_1 a_2 \dots a_k \in \Sigma^*$ $(k \geq 0)$ е дума и $a_i \in \Sigma$ $(1 \leq i \leq k)$ са букви. Ако $\delta_L^*(a_1 a_2 \dots a_k)$ и $\delta_R^*(a_k a_{k-1} \dots a_1)$ са дефинирани, то можем да получим двата пътя:

$$\pi_L = l_0 \to^{a_1} l_1 \to^{a_2} \dots l_{k-1} \to^{a_k} l_k$$

$$\pi_R = r_0 \leftarrow^{a_1} r_1 \leftarrow^{a_2} \dots r_{k-1} \leftarrow^{a_k} r_k$$

Където π_L и π_R са пътища в съответно левия и десния автомат и думата w се разпознава от $\mathcal{A}_L$ в посока от ляво на дясно, а от $\mathcal{A}_R$, съответно от дясно на ляво. Ако за всички тройки $\langle l_{i-1}, a_i, r_i \rangle$, изходната функция $\psi(l_{i-1}, a_i, r_i)$ е дефинирана, то двойката пътища $\langle \pi_L, \pi_R \rangle$ наричаме *успешно изпълнение* на $\mathcal{B}$ с етикет $w = a_1 a_2 \dots a_k$ и *изход*

$$\mathcal{O}_\mathcal{B}(w) = \psi(l_0, a_1, r_1) \cdot \psi(l_1, a_2, r_2) \cdot \dots \cdot \psi(l_{k-1}, a_k, r_k)$$

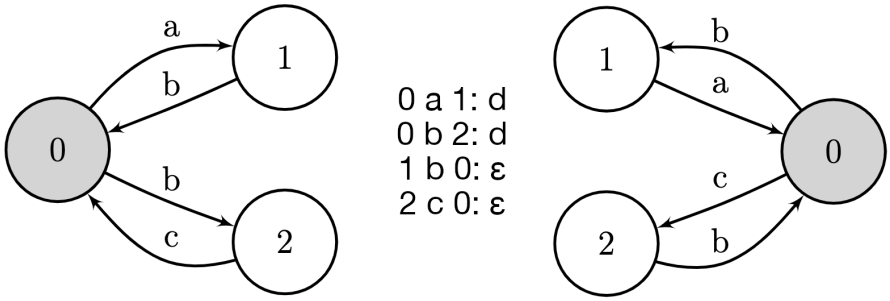
$\mathcal{O}_\mathcal{B}$ наричаме *изходна функция на бимашината* и казваме, че бимашината *превежда* w в m, ако $\mathcal{O}_\mathcal{B}(w) = m$, където m е резултат от конкатенацията на всички $\psi(l_{i-1}, a_i, r_i)$ $(1 \leq i \leq k)$.

Бимашината чете входната дума и за всеки символ извежда дума над азбуката си. На всяка стъпка изведената от изходната функция ψ дума зависи от входния символ и двете състояния в които биха преминали левият и десният автомат, четейки входа съответно от ляво на дястно и от дясно на ляво. Крайният резултат е конкатенацията на всички така изведени думи.

Дефиниция 2.19. Нека $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ е бимашина. *Разширената изходна функция* ψ^* дефинираме индуктивно:

- $\psi^*(l, \epsilon, r) = \epsilon$ за всяко $l \in Q_L, r \in Q_R$
- $\psi^*(l, wa, r) = \psi^*(l, w, \delta_R(r, a)) \cdot \psi(\delta_L^*(l, w), a, r)$, за $l \in Q_L, r \in Q_R, w \in \Sigma^*, a \in \Sigma$

Пример 2.6. *(Бимашина и изпълнение)* Нека разлгледаме бимашината на Фигура 2.4. Задаваме входна дума *bcabbc*, което води до следното изпълнение на левия и десния автомат съответно.



Фигура 2.4: Бимашина представяща $\{\langle ab, d\rangle, \langle bc, d\rangle\}^*$

$$\pi_L = 0 \to^b 2 \to^c 0 \to^a 1 \to^b 0 \to^b 2 \to^c 0$$
$$\pi_R = 0 \leftarrow^b 2 \leftarrow^c 0 \leftarrow^a 1 \leftarrow^b 0 \leftarrow^b 2 \leftarrow^c 0$$

Изходната функция на бимашината $\mathcal{O}_\mathcal{B}$ прилагаме както следва:

$$\psi(0,b,2)\cdot\psi(2,c,0)\cdot\psi(0,a,1)\cdot\psi(1,b,0)\cdot\psi(0,b,2)\cdot\psi(2,c,0) = d\cdot\epsilon\cdot d\cdot\epsilon\cdot d\cdot\epsilon = ddd$$

Дефиниция 2.20. Бимашина $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi\rangle$ наричаме *тотална*, ако функциите на прехода $\delta_L : Q_L \times \Sigma \to Q_L$ и $\delta_R : Q_R \times \Sigma \to Q_R$ на левия и десния автомат съответно, както и функцията на изхода $\psi : (Q_L \times \Sigma \times Q_R) \to \Sigma^*$ са *тотални*.

Теорема 2.4. Класическите бимашини са еквивалентни по изразителност на регулярните функции. [4] [6]

3 Правила за заместване

За двоичните стрингови релации можем да си мислим като множество от преводи, като на пример за двойката $\langle u, v \rangle$ казваме, че думата u се превежда като v.

Дефиниция 3.1. *Правило за заместване* представяме във вида

$$E \to \beta$$

където E е регулярен израз над крайна азбука Σ, а $\beta \in \Sigma^*$ е дума. *Приложение на правилото върху текст* $t \in \Sigma^*$ представлява заместването на поднизовете на t, които са в езика $L(E)$ с β.

$$a_1 a_2 \ldots \overbrace{\underbrace{a_i \ldots a_{i+k}}_{\beta}}^{\in L(E)} \ldots \overbrace{\underbrace{a_j \ldots a_{j+l}}_{\beta}}^{\in L(E)} \ldots a_n$$

Фигура 3.1: Приложение на правило на заместване

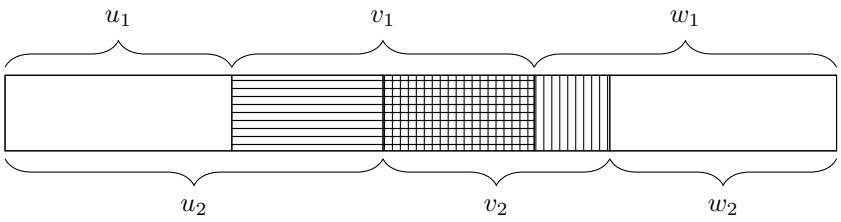
Правила за заместване можем да представим като *регулярни стрингови релации*, използвайки единствено алгебрата на регулярните езици и релации. Те се реализират програмно чрез крайни преобразуватели. [1]

Дефиниция 3.2. Нека разгледаме текст $t \in \Sigma^*$. *Контекст на заместване* наричаме тройка $\langle u, v, w \rangle$, където $t = uvw$. Също така u и w наричаме съответно *префикс* и *суфикс* на контекста, докато v наричаме *фокус*.

Пример 3.1. Нека разгледаме правилото `ab|bc` $\to d$. След приложението му над текста abb, ще получим db, като $\langle \epsilon, ab, b \rangle$ е единственият контекст на заместване. Приложението на правилото над $abbacbca$ ще доведе до $dbacda$ с два контекста на заместване $\langle \epsilon, ab, bacbca \rangle$ и $\langle abbac, bc, a \rangle$.

3.1 Разрешаване на многозначности

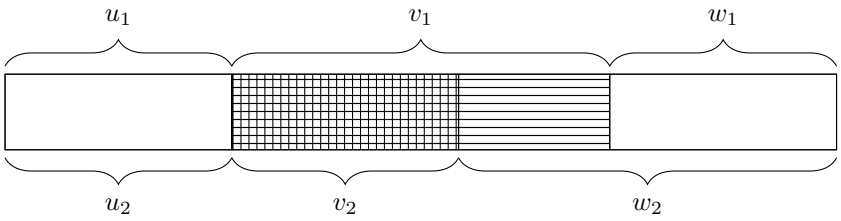
При прилагане на правило за заместване могат да възникнат многозначности в случаите, в които два контекста имат застъпващи се фокуси.



Фигура 3.2: Застъпващи се контексти

Дефиниция 3.3. Два контекста на заместване $\langle u_1, v_1, w_1\rangle$ и $\langle u_2, v_2, w_2\rangle$ за даден текст t *се застъпват*, ако $u_1 < u_2 < u_1v_1$. Израза $u_1 < u_2$ четем като u_1 е *префикс* на u_2.

Пример 3.2. Нека разгледаме правилото `ab|bc` $\rightarrow d$) приложено над текста $t = aabcb$. Получаваме контекстите $\langle a, ab, cb\rangle$ и $\langle aa, bc, b\rangle$, които очевидно се застъпват и съответно стигаме до две различни валидни заместания $adcb$ и $aadb$.



Фигура 3.3: Застъпващи се контексти с еднакво начало

Пример 3.3. Друг вид многозначност може да получим, когато фокусите на два контекста имат еднакво начало. Например, ако приложим правилото $a+ \rightarrow d$ над тескт $t = aa$, може получим превода dd, на който отговарят контекстите $\langle a, a, \epsilon\rangle$ и $\langle \epsilon, a, a\rangle$, и d с контекст $\langle \epsilon, aa, \epsilon\rangle$.

Дефиниция 3.4. Въвеждаме следните оператори над множества от контексти:

$$AFTER(A, B) = \{\langle u, v, w\rangle \in A \mid \forall\langle u', v', w'\rangle \in B : u' \cdot v' \leq u \wedge u' < u\}$$

$AFTER$ избира измежду всички контексти в множеството A тези, в които фокусът v започва след всички фокуси в множеството B.

$$LEFTMOST(A) = \{\langle u, v, w\rangle \in A \mid \forall\langle u', v', w'\rangle \in A : u \leq u'\}$$

$LEFTMOST$ избира измежду всички контексти в A, тези с чийто фокус се намира възможно най в ляво. Може да имаме повече от един такъв контекст.

$$LONGEST(A) = \{\langle u, v, w\rangle \in A \mid \forall\langle u', v', w'\rangle \in A : u \neq u' \vee v' \leq v\}$$

Измежду контекстите, чиито фокуси започват от една и съща позиция, $LONGEST$ избира тези, които имат най-дълъг фокус.

Дефиниция 3.5. Въвеждаме оператора $LML(A)$ (leftmost-longest), чрез който ще елиминираме многозначностите. По дадено множество от контексти A, $LML(A)$ избира най-левите, най-дълги измежду тях.

$$LML(A) := \bigcup_{i=0}^{\infty} C_i$$

Където междинните множества C_i строим по индукция:

$$C_0 = \emptyset$$
$$C_{i+1} = C_i \cup LONGEST(LEFTMOST(AFTER(A, C_i)))$$

$LML(A)$ е крайно множество, защото A е крайно, т.е. след дадем момент редицата ще престане да нараства.

Твърдение 3.1. Нека $t \in \Sigma^*$ е текст и A е множество от контексти в t. $LML(A)$ съдържа само незастъпващи се контексти.

Доказателство. Нека $LML(A)$ съдържа застъпващите се контексти $\langle u_1, v_1, w_1\rangle$ и $\langle u_2, v_2, w_2\rangle$ т.е. $u_1 < u_2 < u_1v_1$. Също така $\langle u_1, v_1, w_1\rangle \in C_{i_1}$ и $\langle u_2, v_2, w_2\rangle \in C_{i_2}$. Нека допуснем, че $i_1 \geq i_2$. В такъв случай:

$$\langle u_2, v_2, w_2\rangle \in C_{i_2} = C_{i_2-1} \cup LONGEST(LEFTMOST(AFTER(A, C_{i_2-1})))$$

След като $\langle u_2, v_2, w_2\rangle \notin C_{i_2-1}$, то $\langle u_2, v_2, w_2\rangle \in LEFTMOST((AFTER(A, C_{i_2-1}))$. От допускането, че $i_1 \geq i_2$ следва, че $\langle u_1, v_1, w_1\rangle \in AFTER(A, C_{i_2-1})$, което е в противоречие с дефиницията на $LEFTMOST$ и следователно $i_1 < i_2$.
След като $i_1 < i_2$, то $\langle u_2, v_2, w_2\rangle \notin C_i$ за $i \leq i_1$. При $i > i_1$, $\langle u_1, v_1, w_1\rangle \in C_i$, значи $\langle u_2, v_2, w_2\rangle \in AFTER(A, C_i)$, което не е възможно спрямо дефиницията на $AFTER$ т.е. $\langle u_2, v_2, w_2\rangle \notin C_{i+1}$. Това противоречи с факта, че $\langle u_2, v_2, w_2\rangle \in C_{i_2}$ $(i_1 < i_2)$, следователно $LML(A)$ не съдържа застъпващи се контексти. □

Дефиниция 3.6. Нека $T \subseteq \Sigma^* \times \Sigma^*$ е релация и $t \in \Sigma^*$ е текст. С $A_{dom(T)}$ бележим множеството на всички поднизове в t, които са в $dom(T)$. Нека

$$t = u_1 v_1 x_2 v_2 \dots x_k v_k w_k$$

е каноничното представяне на t за множеството $LML(A_{dom(T)})$. Тогава t' е *заместването* на t с T под стратегията най-ляво-най-дълго срещане

$$t = u_1 v'_1 x_2 v'_2 \dots x_k v'_k w_k$$

и $\langle v, v' \rangle \in T$ за $1 \leq i \leq k$. С $R^{LML}(T)$ бележим релацията на заместване под стратегията най-ляво-най-дълго срещане за T. Тя съдържа всички двойки $\langle t, t' \rangle \in \Sigma^* \times \Sigma^*$, така че t' е заместване на t с T под стратегията най-ляво-най-дълго срещане.

Следствие 3.1. Нека $T : \Sigma^+ \to \Sigma^*$ е функция, тогава релацията на заместване $R^{LML}(T)$ е функционална.

Пример 3.4. Нека разгледаме правилото `ab|bc` $\to d$ съответстващо на релацията $T = \{\langle ab, d \rangle, \langle bc, d \rangle\}$, приложено над текста $t = aabcbab$. Получаваме контекстите $\langle a, ab, cbab \rangle$, $\langle aa, bc, bab \rangle$ и $\langle aabcb, ab, \epsilon \rangle$. Очевидно първите два се застъпват, но стратегията най-ляво-най-дълго срещане определя първият и третият за валидни. Резултатът от заместането е $R^{LML}(T)(aabcbab) = adcbd$.

3.2 Регулярни релации за лексически анализ

За да разбием даден текст на редица от тоукъни, е нужно да представим тези тоукъни под формата на регулярни изрази. Това представяне наричаме *лексическа граматика.*

Пример 3.5. *Лексическа граматика и извличане на тоукъни.*

Token	**Description**
Number	`[0-9]+(\.[0-9]+)?`
Operator	`+|-|*|/`
Equal	`=`

Фигура 3.4: Лексическа граматика на аритметичен израз

Символите на думата *15+9-3=21* се групират в следната редица от тоукъни спрямо граматиката: `15`, `+`, `9`, `-` , `3`, `=`, `21`, докато за *+-**3232*, получаваме `+`, `-`, `*`, `*`, `3232`.

Регулярните релации намират своето приложение в множество домейни, като лексическият анализ е един от тях [3]. Идеята е да сведем процеса на токенизация до заместване на думи в текст, като от граматиката построим релация на заместване (под стратегията най-ляво-най-дълго срещане). Тази релация представяме програмни като краен преобразувател, който поставя маркер след всеки разпознат тоукън от лексическата граматика. В последствие от преобразувателят строим еквивалентна бимашина.

$$E \to \ldots \texttt{EoT}$$

Тази нотация представлява правило на заместване, което за дума в езика $L(E)$ на регулярния израз, добавя маркер в края ѝ (*end-of-token*). Със символа $\ldots$ означаваме, че разпознатата дума се замества със себе си (т.е. входът остава непроменен). Формално, такъв тип релаци по зададен регулярен получаваме израз като конкатенираме идентитета на неговия език със синглетон, който представлява заместване на празната дума с маркер за край на тоукън (end-of-token).

$$Id(L(E)) \cdot \{\langle \epsilon, \texttt{EoT} \rangle\}$$

Граматиката от Пример 3.5, представяме по следния начин:

1. Number: `[0-9]+` $\rightarrow$ ... `EoT`
2. Operator: `+|-|*|/` $\rightarrow$ ... `EoT`
3. Equal: `=` $\rightarrow$ ... `EoT`

Всяко правило в граматиката е регулярна релация като *релацията за лексически анализ* получаваме като обединим релациите, съответстващи на тези правила и от това обединение построим релация на заместване под стратегията *най-ляво-най-дълго* срещане.

$$R^{LML}(R_1 \cup R_2 \cup \ldots \cup R_n)$$

Както установихме, $R^{LML}(T)$ е функционална, ако T е функция. Тъй като една коректно дефинирана лексическа граматика не може да има две правила, които разпознават една и съща дума, то условието е изпълнено.
Нека се върнем на граматиката от Пример 3.5. Релацията за лексически анализ представяме по следния начин:

$$R := R^{LML}(R_{Num} \cup R_{Op} \cup R_{Eq})$$

$$R(\textit{42-15*5}) = \texttt{"42 EoT - EoT 15 EoT * EoT 5 EoT"}$$

Работата на лексическият анализатор не се изчерпва с извличането на токъните от входния текст. Освен съдържанието му, често се нуждаем от информация и за неговия тип, както и позиция в текста. Типът е необходим при извършването на синтактичният анализ, докато позицията е полезна, когато съобщаваме за грешки във входния текст било то по време на синтактичния, или лексическия анализ.
Нека променим правилото на заместване, така че да включва и типа.

$$E \rightarrow \texttt{Type SoT} \ldots \texttt{EoT}$$

`Type` е индикатор, който носи информация за това кое правило на заместване е било приложено, докато `SoT` маркира началото на лексемата. Релациите за тези правила получаваме по следния начин:

$$\{\langle \epsilon, \texttt{Type SoT} \rangle\} \cdot Id(L(E)) \cdot \{\langle \epsilon, \texttt{EoT} \rangle\}$$

Съответно граматиката от Пример 3.5 придобива следния вид:

1. Number: `@1 SoT [0-9]+` $\rightarrow$ ... `EoT`

2. Operator: `@2 SoT +|-|*|/` $\rightarrow$ `...EoT`

3. Equal: `@3 SoT =` $\rightarrow$ `...EoT`

Релацията за лексически анализ R получаваме по същия начин, но изходната дума вече съдържа повече информация.

$$R(999+1) =$$

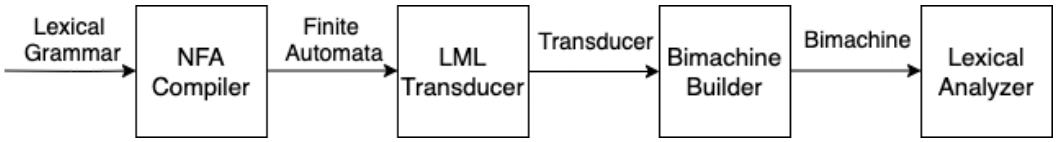
```
"@1 SoT 999 EoT @2 SoT + EoT @1 SoT 1 EoT @3 SoT = EoT"
```

4 Реализация

4.1 Дизайн

Реализацията на генератор за лексически анализ чрез бимашина е разделена на четири етапа:

1. Конструкция на краен автомат по регулярен израз.
2. Конструкция на преобразувател по стратегията най-ляво-най-дълго срещане.
3. Превръщане на функционален преобразувател в еквивалентна бимашина.
4. Алгоритъм за лексически анализ чрез симулация на бимашина.



В първата част ще представим разширен синтаксис на регулярен израз и разработка на recursive descent парсър, чрез който строим еквивалентния краен автомат. Във втората част ще разгледаме как се строи преобразувател от правила на заместване по стратегията най-ляво-най-дълго срещане и съответно как се строи бимашина по такъв преобразувател. Конструкцията на преобразувателя се съставя изцяло от операции над регулярни езици и релации. В третата част ще видим как се осъществява лексически анализ, чрез получената бимашина.

4.2 От регулярен израз към краен автомат

Регулярните изрази датират още от 50'те години на 20'ти век и са интегрирани в стандартните библиотеки на всички модерни езици за програмиране. Съществуват различни стандарти за представянето им, като тук ще ползваме синтаксис сходен с този на регулярните изрази в Perl. На Фигура 4.2 са представени базовите синтактични единици, чрез които конструираме регулярните изрази.

Пример 4.1. Нека разгледаме няколко примерни регулярни израза спрямо описания синтаксис.

Израз	Пояснение
`ab`	конкатенация на символите 'a' и 'b'
`|`	обединение (`A|B` - думата се разпознава от `A` или `B`)
`*`	0 или повече срещания (Звезда на Килни)
`+`	1 или повече срещания
`?`	0 или 1 срещане
`.`	всеки символ от азбуката
`(...)`	начало и край на група (разпознава израза в скобите)
`[...]`	множество от символи (`[0-9]` разпознава цифрите от 0 до 9)
`[^...]`	отрицание (разпознава символите, които не са в множеството)
`{n}`	точно n на брой срещания
`{n,}`	поне n на брой срещания
`{n,m}`	от n до m на брой срещания
`\`	буквална интерпретация на символ (`\*` разпознава '*')

Фигура 4.1: Синтаксис на регулярен израз

- `(a|b)*c` - разпознава думите, които се състоят от 0 или повече символи 'a' и 'b' и завършват на 'c' като на пример: *ac, bc, abbabac, bbac, c, ...*
- `.*true.*` - думите, които съдържат "true": *abctrue, 1true2_ , true, ...*
- `[a-zA-Z0-9@]+` - разпознава думите, които се състоят единствено от латински букви, цифри и '@': *z, 1b@, AbC123, 404, ccC, ...*
- `[^ \t\r\n]` - разпознава всички символи, които не са интервал, табулация или нов ред.

За да построим краен автомат по регулярен израз е нужно първо да дефинираме граматическата му структура. Въвеждаме следната безконтекстна граматика на регулярен израз:

$\langle expr \rangle ::= \langle term \rangle$
$\quad | \quad \langle term \rangle$ '|' $\langle expr \rangle$

$\langle term \rangle ::= \langle factor \rangle$
$\quad | \quad \langle factor \rangle \langle term \rangle$

⟨*factor*⟩ ::= ⟨*atom*⟩
  | ⟨*atom*⟩ ⟨*meta-char*⟩
  | ⟨*atom*⟩ '{' ⟨*char-count*⟩ '}'

⟨*atom*⟩ ::= ⟨*char*⟩
  | '.'
  | '(' ⟨*expr*⟩ ')'
  | '[' ⟨*char-class*⟩ ']'
  | '[' '^' ⟨*char-class*⟩ ']'

⟨*char-class*⟩ ::= ⟨*char-class-item*⟩
  | ⟨*char-class-item*⟩ ⟨*char-class*⟩

⟨*char-class-item*⟩ ::= ⟨*char*⟩
  | ⟨*char*⟩ '' ⟨*char*⟩

⟨*char-count*⟩ ::= ⟨*integer*⟩
  | ⟨*integer*⟩ ','
  | ⟨*integer*⟩ ',' ⟨*integer*⟩

⟨*integer*⟩ ::= ⟨*digit*⟩
  | ⟨*digit*⟩ ⟨*integer*⟩

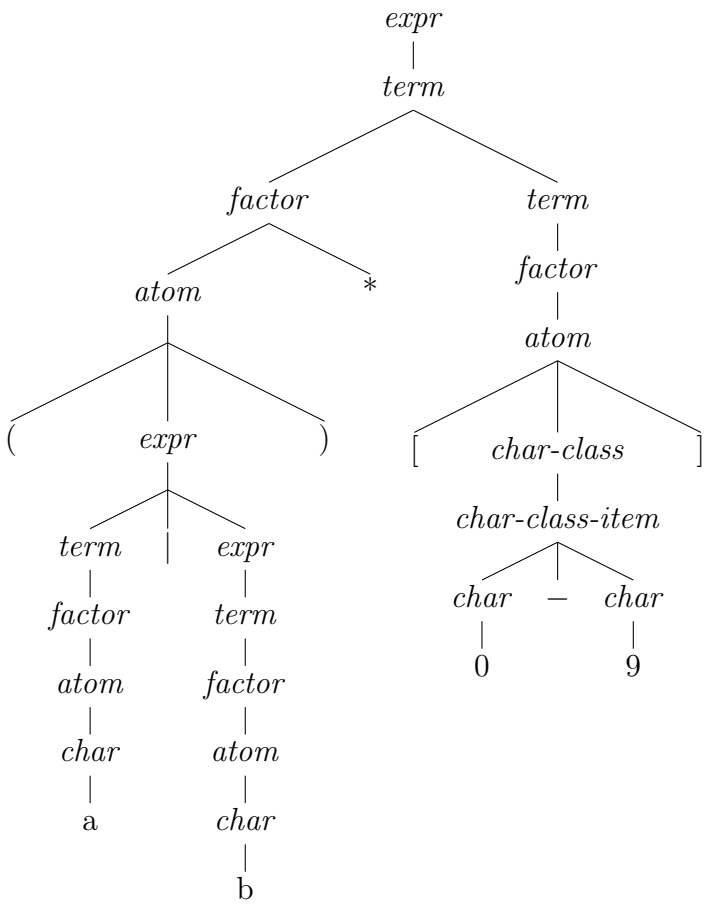
⟨*char*⟩ ::= *anyCharExceptMeta*
  | '\' ⟨*any-char*⟩

⟨*any-char*⟩ ::= *alphabet*

⟨*meta-char*⟩ ::= '?'
  | '*'
  | '+'

⟨*digit*⟩ ::= '0' | '1' | . . . | '9'

Следвайки тази граматика за всеки коректно дефиниран регулярен израз извличаме дърво на извод, по което чрез обхождане в дълбочина и строим недетерминиран краен автомат. Представяме автоматите и реализираме операциите



Фигура 4.2: Дърво на извод за регулярния израз `(a|b)*[0-9]`, който разпознава думите, започващи с 0 или повече на брой символи 'a' и 'b', които завършват с цифра, на пример *ab3, bbbb4, 9, aa1, baab4* и т.н.

по между им (конкатенация, обединение, звезда...) спрямо алгоритъма на Томпсън [5]. На Фигура 4.2 е изобразено дърво на извод на регулярен израз спрямо граматиката.

Извличането на дърво на извод по зададен регулярен израз реализираме чрез *рекурсивно спускане (recursive descent parser)*. Той спада към класа на типа парсъри, които строят дървото на извод *"от горе надолу"(top-down)*, т.е. от корена към листата, започвайки от аксиомата на граматиката (нашия случай аксиомата е нетерминалът *expr*). Граматически правила се прилагат от ляво на дясно. Регулярен израз е коректно дефиниран, ако по него може да се построи дърво на извод.

Особеност на *top-down* парсърите е, че работят с ограничен клас от безконтекстни граматики. Ако граматиката съдържа *лява рекурсия*, то е възможно процедурата да изпадне в безкраен цикъл поради факта, че правилата се прилагат от ляво на дясно. Нека разгледаме следния пример за директна рекурсия:

$$A \to A\alpha \mid \beta$$

Символът A е нетерминал, докато α и β са думи, съдържащи терминали и нетерминали, които обаче не започват с A. Тъй като прилагаме правилата от ляво на дясно, то A ще опита първоначално да изведе A и така ще изпаднем в безкраен цикъл. Решението тук е да преобразуваме правилото до еквивалентната му дясно-рекурсивна форма.

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$

Граматиката на регулярен израз не съдържа лява рекурсия, съответно можем по нея да реализираме *recursive descent* парсър. Идеята е следната:

- За всеки нетерминал (*expr, term, factor...*) имплементираме отделен метод.
- Построяването на дървото започва с извикването на аксиомата на граматиката (*Expr()*).
- Използваме методите *Peek()*, за да видим следващия непрочетен символ от входа и *Eat(char ch)*, който сравнява следващия следващия символ с *ch* и преминава с една позиция напред. *Next()* преминава към следващата позиция във входната дума без значение от символа, т.е. *Eat(Peek())*.

Нека разгледаме реализацията на правилото за аксиомата на граматиката.

⟨*expr*⟩ ::= ⟨*term*⟩
  | ⟨*term*⟩ '|' ⟨*expr*⟩

Нетерминалът *expr* има два извода, като и двата започват с нетерминала *term*. След като приложим правилото *term*, проверяваме дали не сме стигнали до края на входа и в случай, че следващият символ е операторът за обединението, то местим входната дума с една позиция, прилагаме правилото *expr* и връщаме обединението на резултатите.

```
1  Fsa Expr()
2  {
3    var term = Term();
4    if (HasMoreChars() && Peek() == '|') {
5      Eat('|');
6      return term.Union(Expr());
7    }
8    return term;
9  }
```

Важно е да уточним, че типът `Fsa` представлява краен автомат т.е. всеки метод, отговарящ на нетерминал в граматиката връща краен автомат. По този начин финалният автомат, съответстващ на регулярния израз се строи директно по време на получаването на дървото на извод. Нека разгледаме метода за нетерминала *char*, който винаги извежда терминален символ, т.е. сме стигнали до листо на дървото.

```
1  Fsa Char()
2  {
3    if (Peek() == '\') {
4      Eat('\');
5      var ch = Next();
6      if (!alphabet.Contains(ch))
7        throw new Exception($"Invalid char {ch}");
8      return FsaBuilder.FromSymbol(ch);
9    }
10   var ch = Next();
11   if (metaChars.Contains(ch))
12     throw new Exception($"Unescaped meta char {ch}");
13   return FsaBuilder.FromSymbol(ch);
14 }
```

Ако сме стингали до символа '\', последвалият го символ третираме като буква, независимо дали се използва като оператор. Методът връща автомат, който разпознава единствено прочетения символ.

4.3 Конструкция на преобразувател по стратегията най-ляво-най-дълго срещане

Алгоритъм за построяване на краен преобразувател по правила на заместване е представен в [1] и развит в последвалите разработки [3], [2] и други. Базирайки се на тези статии, ще представим метод за имплементация на краен преобразувател по стратегията най-ляво-най-дълго срещане използвайки единствено алгебрата на регулярните езици и релации.

Дефинираме метода *Fst ToLmlRewriter(Fst fst, ISet<char> alphabet)*, който по даден краен преобразувател и азбука, строи преобразувател по стратегията най-ляво-най-дълго срещане. Този метод извежда преобразувател, който представя релацията $R^{LML}(T)$. Алгоритъмът се състои от четири стъпки като на всяка стъпка строим краен преобразувател и резултатът е композицията на тези преобразуватели в реда, в който са дефинирани.

1. *Първоначално срещане.* Строим преобразувател, който поставя маркер (`cb`) в началото на всяко срещане на поддума (фокус) от входния текст, която е в домейна на входния преобразувател. Тази стъпка е еквивалентна на оператора $AFTER$ и идентифицира всички контексти на заместване в текста независимо от това дали се застъпват, или са с възможно най-дълги фокуси.

2. *От ляво на дясно.* Този преобразувател получава текста с маркерите от първата стъпка и поставя в началото и края на всеки контекст маркерите `lb` и `rb`, като `cb` се замества с `lb`. Между маркерите за начало и край, не може да съществуват други маркери. Това може да доведе до повече от един изходен текст. Тази стъпка отговаря на оператора $LEFTMOST$ и осигурява, че измежду контекстите със застъпващи се фокуси, ще останат тези с най-ляв фокус.

3. *Най-дълго-срещане.* Преобразувателят получава изхода от втората съпка и измежду маркираните текстове съдържащи контексти с еднакво начало, извежда такъв, в който фокусите са най-дълги. Тази стъпка отговаря на оператора $LONGEST$.

4. *Заместване.* В тази стъпка получаваме изходния текст от стъпка 3, в който поддумите в домейна на преобразувателя са маркирани с `lb` и `rb` под стратегията най-ляво-най-дълго срещане. Този преобразувател извършва заместването на тези поддуми чрез входния преобразувател и премахва всички маркери.

Пример 4.2. Имаме правилата на заместване `ab+` $\to x$ и `bc` $\to y$, съответстващи на релациите R_1, R_2. Входният преобразувател T представлява обединението: $T := T_{R_1} \cup T_{R_2}$. Входният текст е *abcabbdbc* и маркерите `cb`, `lb` и `rb` представяме съответно чрез символите '!', '<' и '>'.

1. Маркираме началото на всички фокуси в думата: *!a!bc!abbd!bc*
2. Идентифицираме най-левите незастъпващи се контексти, което води до следните възможно декомпозиции: $<ab>c<ab>bd<bc>$ и $<ab>c<abb>d<bc>$
3. Прилагаме преобразвателят за най-дълго срещане и получаваме $<ab>c<abb>d<bc>$
4. Заместваме поддумите спрямо входния преобразувател и получаваме *xcxdy*

За да имплементираме тези преобразуватели, нека дефинираме следните оператори над регулярни езици и релации представени в [1]:
Intro, IntroX, Ingore ...

4.4 От краен преобразувател към бимашина

Преобразувателят, построен по стратегията най-ляво-най-дълго срещане е функционален, по него можем да построим еквивалентна в бимашина. Ще имплементираме алгоритъма представен в [6].

Първата стъпка е да детерминираме подлежащия огледален автомат на входния преобразувател. Припомняме, че огледален на даден автомат е такъв, на който сме "обърнали срелките"на преходите.

```
1  Bimachine ToBimachine(this Fst fst, ISet<char> alphabet)
2  {
3      var fstTransGroupedByTarget = fst.Transitions
4          .GroupBy(tr => tr.To)
5          .ToDictionary(
6              g => g.Key,
7              g => g.Select(tr => (In: tr.In, To: tr.From)));
8
9      var rightSStates = new List<ISet<int>> { fst.Final.ToHashSet() };
10     var rightTrans = new Dictionary<(int From, char Label), int>();
11
12     for (int n = 0; n < rightSStates.Count; n++) {
13         var symbolToSStates = rightSStates[n]
14             .Where(st => fstTransGroupedByTarget.ContainsKey(st))
15             .SelectMany(st => fstTransGroupedByTarget[st])
16             .GroupBy(tr => tr.In, tr => tr.To)
17             .ToDictionary(g => g.Key, g => g.ToHashSet());
```

```
18
19          foreach (var (symbol, subsetState) in symbolToSStates)
20              if (!rightSStates.Any(rs => rs.SetEquals(subsetState)))
21                  rightSStates.Add(subsetState);
22
23          foreach (var (label, targetSState) in symbolToSStates)
24              rightTrans.Add(
25                  (n, label.Single()),
26                  rightSStates.FindIndex(
27                      ss => ss.SetEquals(targetSState)));
28  }
```

На редове 3-7 групираме преходите на подлежащия автомат по състоянията, **към които**, има преходи. На пример, ако имаме двата прехода $\langle p_1, a, c, q\rangle$ и $\langle p_2, b, d, q\rangle$, то ще получим $\langle q, \{\langle a, p_1\rangle, \langle b, p_2\rangle\}\rangle$. Използвайки тази структура, строим множествата на състоянията и преходите на детерминирания подлежащ огледален автомат (редове 9-29). Тези множества принадлежат на десния автомат на бимашината.

...

4.5 Лексически анализ чрез бимашина

В тази част ще съединим частите на конструкцията от регулярните изрази до построяването на бимашина, както и ще представим алгоритъм за лексически анализ чрез симулация на бимашина. Крайният резултат е програмен интерфейс, който използваме по следния начин:

```
1  var lexer = Lexer.Create(new[] {
2      new Rule("[0-9]+\\.?[0-9]*", "NUM"),
3      new Rule("[+*/-]", "OP"),
4      new Rule("=", "EQ"),
5  });
6  lexer.Input = new InputStream("3.14+1.86=5");
7
8  foreach (Token token in lexer.GetNextToken())
9      Console.WriteLine(token);
```

Тази програма ще генерира лексически анализатор, извърши токенизацията на аритметичен израз и изведе следният резултат

```
1  [@0,0:3='3.14',<NUM>]
2  [@1,4:4='+',<OP>]
3  [@2,5:8='1.86',<NUM>]
4  [@3,9:9='=',<EQ>]
5  [@4,10:10='5',<NUM>]
```

Разглеждаме процедурата по създаване на лексическия анализатор.

```
1  const char SoT = '\u0002';
2  const char EoT = '\u0003';
3
4  static Lexer Create(IList<Rule> grammar)
5  {
6      var tokenFsts = new List<Fst>();
7
8      for (int i = 0; i < grammar.Count; i++) {
9          var ruleFsa = new RegExp(grammar[i].Pattern).Automaton;
10         var ruleFst = FstBuilder.FromWordPair("", $"{i}{SoT}")
11             .Concat(ruleFsa.Identity())
12             .Concat(FstBuilder.FromWordPair("", $"{EoT}"));
13         tokenFsts.Add(ruleFst);
14     }
15
16     var unionFst = tokenFsts.Aggregate((u, f) => u.Union(f));
17     var alphabet = unionFst.Transitions
18         .Where(t => !string.IsNullOrEmpty(t.In))
19         .Select(t => t.In.Single())
20         .ToHashSet();
21
22     var lmlFst = unionFst.ToLmlRewriter(alphabet);
23     var bm = lmlFst.ToBimachine(alphabet);
24
25     return new Lexer(bm, grammar);
26 }
```

На редове 1-2 декларираме символите, с които ще означаваме началото и края на лексемите. На 6-14 обхождаме правилата в граматиката и по регулярния израз на всяко строим краен автомат. В последствие по този автомат получаваме преобразувател, който за всяка дума в езика му ще изведе думата с прикрепяйки индекса на правилото и маркерите за начало и край. На 16-20 обединяваме тези преобразуватели и определяме азбуката като вземем уникалните символи по горната лента на получения преобразувател. От него на ред 22 строим такъв под стратегията "най-ляво-най-дълго"срещане, който превръщаме в еквивалентна бимашина (ред 23).

```
1  class Dfsa
2  {
3      public IList<int> States { get; set; }
4      public int Initial { get; set; }
5      public IList<int> Final { get; set; }
6      public IDictionary<(int From, char Label), int>
7          Transitions { get; set; }
```

```
8 }
```

Така представяме детерминиран краен автомат, като функциата на прехода реализираме като двойки ключ-стойност, където ключът е двойка от състояние и символ, а стойността е състоянието, в което попадаме след този преход.

```
1 class Bimachine
2 {
3     public Dfsa Forward { get; set; }
4     public Dfsa Reverse { get; set; }
5     public IDictionary<(int Lstate, char Symbol, int Rstate), string>
6         Output { get; private set; }
7 }
```

Аналогично за бимашината имаме два автомата. "Forward"отговаря на левия автомат, който сканира текста от ляво на дясно и "Reverse който сканира текста от дясно на ляво. Функцията на изхода са двойки ключ-стойност, където ключът е тройка, състояща се от състояние на левия автомат, входен символ и състояние на десния автомат, стойността е изведената дума. Вече сме готови да представим метода за токенизация.

```
1 class Lexer
2 {
3     char SoT = '\u0002';
4     char EoT = '\u0003';
5     IList<Rule> grammar;
6
7     Lexer(Bimachine bm, IList<Rule> grammar)
8     {
9         this.Bm = bm;
10        this.grammar = grammar;
11    }
12
13    public Bimachine Bm { get; set; }
14    public InputStream Input { get; set; }
```

Лексическият анализатор пази референции към граматиката и съответстващата ѝ бимашина. Типът *InputStream* е абстракция на входния текст, който може да се подаде като низ от символи в паметта или зареди от текстов файл.

```
15    public IEnumerable<Token> GetNextToken()
16    {
17        var rPath = Bm.Reverse.ReverseRecognitionPath(Input);
18
19        if (rPath.Count != Input.Size + 1)
20            throw new ArgumentException(
```

```
21                  $"Invalid input symbol. {Input.CharAt(Input.Size -
                        rPath.Count)}");
22
23              var leftState = Bm.Forward.Initial;
24              var token = new StringBuilder();
25              var typeIndex = new StringBuilder();
26              var tokenIndex = 0;
27              var tokenStartPos = 0;
28
29              for (Input.SetToStart(); !Input.IsExhausted; Input.MoveForward
                    ())
30              {
31                  var ch = Input.Peek();
32                  var rightIndex = rPath.Count - 2 - Input.Pos;
33                  var triple = (leftState, ch, rPath[rightIndex]);
34
35                  if (!Bm.Output.ContainsKey(triple))
36                      throw new ArgumentException($"Invalid token '{token.
                            ToString() + ch}'");
37
38                  token.Append(Bm.Output[triple]);
39
40                  if (token[token.Length - 1] == EoT)
41                  {
42                      token.Remove(token.Length - 1, 1);
43                      for (var i = 0; token[i] != SoT; i++)
44                          typeIndex.Append(token[i]);
45                      token.Remove(0, typeIndex.Length + 1);
46
47                      yield return new Token
48                      {
49                          Index = tokenIndex,
50                          Position = (tokenStartPos, Input.Pos),
51                          Text = token.ToString(),
52                          Type = grammar[int.Parse(typeIndex.ToString())].
                                Name
53                      };
54
55                      token.Clear();
56                      typeIndex.Clear();
57                      tokenIndex++;
58                      tokenStartPos = Input.Pos + 1;
59                  }
60
61                  if (!Bm.Forward.Transitions.ContainsKey((leftState, ch)))
62                      throw new ArgumentException($"Invalid input. {ch}");
```

```
63
64                  leftState = Bm.Forward.Transitions[(leftState, ch)];
65              }
66          }
67      }
```

На ред 17 извличаме редицата от състояния получена от симулацията на десния авомат, прочитайки входния текст от дясно на ляво. Ако автоматът не е автоматът не е прочел целия текст, то входът е невалиден връщаме грешка (19-21). Започваме симулацията на левия автомат, заедно с извеждането на лексемите. Намираме се в началното му състояние (23), променливата *token* (24) изпозваме, за да съхраняваме текста на прочетената лексема, *typeIndex* (25) съдържа индекса на правилото в граматиката, чрез който определяме какъв тип е изведената лексема, *tokenIndex* (26) определя коя по ред лексема сме извели, а *tokenStartPos* (27) показва на коя позиция в текста започва лексемата. Четем входа от лявно на дясно и на всеки прочетен символ формираме тройката от ляво, дясно състояние и входен символ (29-33). Ако фукнцията на изхода не е дефинирана за тази тройка, връщаме грешка, в противен случай конкатенираме изходната дума към текста, който сме извели до момента (35-38). Ако последният символ на изведената дума е маркер за край на лемсема (end of token) (40), то сме разпознали лексема и преминаваме към извеждането ѝ. В този момент *token* е низ, който съдържа едновременно идекса на лексемата и нейната стойност (`<index> SoT <text> EoT`). Елиминираме маркера за край, прочитаме символите от началото докато стигнем до `SoT`, което представлява индекса на правилото, съхраняваме ги в *tokenIndex* и елиминираме сегмента, така че в *token* да остане само текста на лексемата (42-45), докато *tokenIndex* съдържа идекса на правилото в граматиката. На редове 47-53 извеждаме обект, който съдържа данните на разпознатата лексема - нейният индекс, позиция, текст и тип. Чрез ключовата дума `yield` указваме, че при следващото извикване на *GetNextToken()*, методът ще продължи изпълнението си, от където е приключил. Подготвяме променливите за разпознаването на следващата лексема (55-58) и продължаваме с итерацията над входния текст, докато не стигнем неговия край, или пък докато левия автомат не може да продължи (61-64).

Литература

[1] Kaplan, Ronald and Kay, Martin. (1994) *Regular models of phonological rule systems.* Computational Linguistics 20(3):331-378

[2] Gerdemann, Dale and van Noord, Gertjan. (1999) *Transducers from rewrite rules with backreferences* EACL '99: Proceedings of the ninth conference on European chapter of the Association for Computational Linguistics June 1999 Pages 126–133

[3] Karttunen, Lauri. (1996) *Directed Replacement.*

[4] Schützenberger, M.-P. (1961) *A remark on finite transducers.* Information and Control, 4:185–196.

[5] Thompson, Ken (1968) *Regular Expression Search Algorithm.* Association for Computing Machinery

[6] Gerdjikov, S., Mihov, S., and Schulz, K. U. (2017) *A simple method for building bimachines from functional finite-state transducers.* Carayol, A. and Nicaud, C., editors, Implementation and Application of Automata, pages 113–125. Springer International Publishing.