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 тип, съдържание 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.

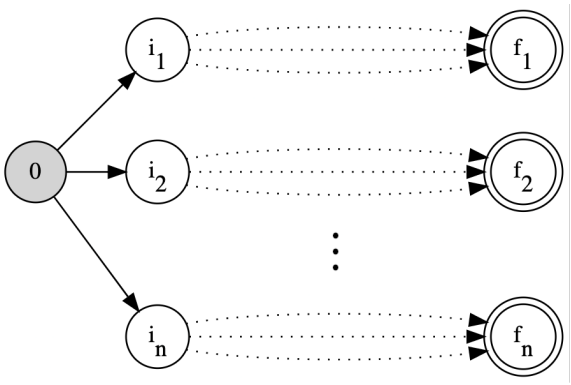
Лексемите се дефинират чрез регулярни изрази. Генератор за лексически анализ получава редица от регулярни изрази като вход и въз основа на тях строи лексически анализатор.

Тази работа представя 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 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:

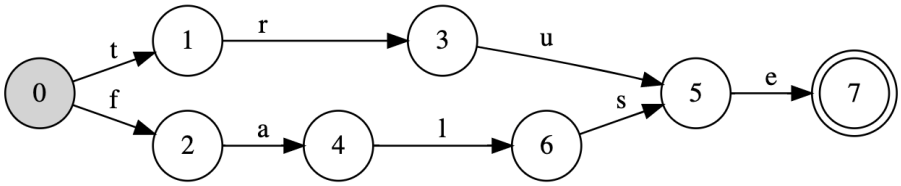
Token	Description
Id	`[a-zA-Z_][a-zA-Z0-9_]*`
Number	`[0-9]+(\.[0-9]+)?`
Boolean	`true|false`
Operator	`=|==|!=|<|<=|>|>=`
If	`if`
Else	`else`
Return	`return`
BraceOpen	`{`
BraceClose	`}`
WS	`[ \t\r\n]+`

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**).

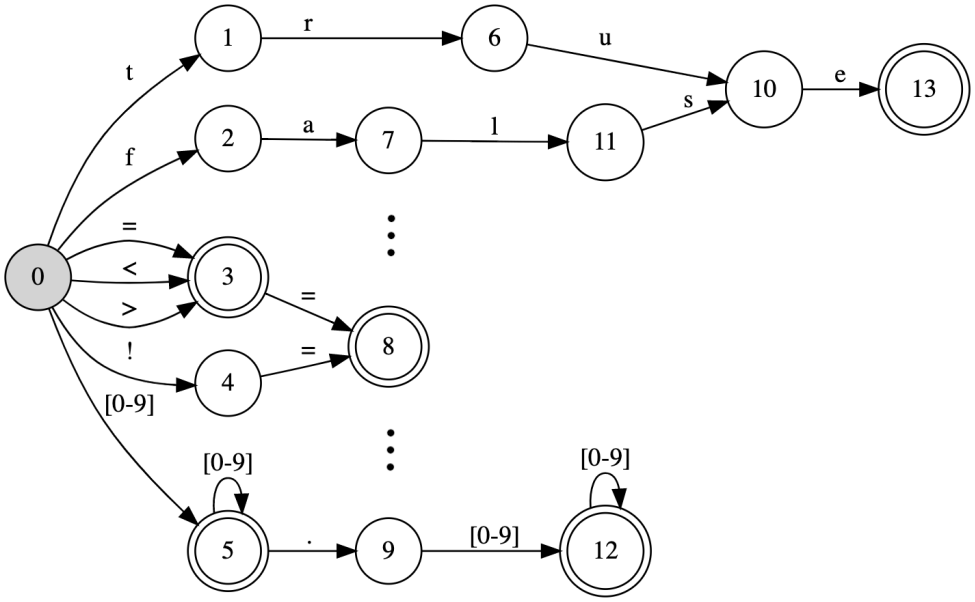
Всяко правило се представя чрез краен автомат, като на пример този за "Boolean"е изобразен на Фигура 1.3.

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.



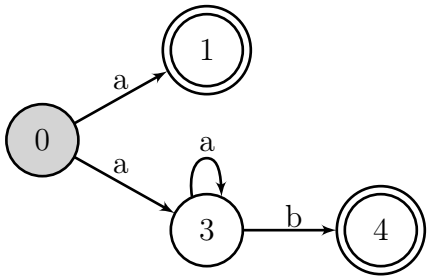
Фигура 1.4: Краен автомат, получен от обединението на автоматите от лексическата граматика. За яснота са изобразени само Boolean, Operator и Number.

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 пример.

Token	Description
A	`a`
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 *"a"*. Правило **B** обхваща думите, съдържащи *'a'* поне веднъж, които завършват с *'b'* като на пример *"ab "aab "aaab"* и т.н. Входният текст *"aabaa"* се разбива на следните лексеми: (*'aab'*, **B**), (*'a'*, **A**), (*'a'*, **A**).

Нека разгледаме случая, в който сканираме текста *"aaaaaaaaaa"*, състоящ се от десет еднакви лексеми - (*'a'*, **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 $q_1 \rightarrow^m q_2$.

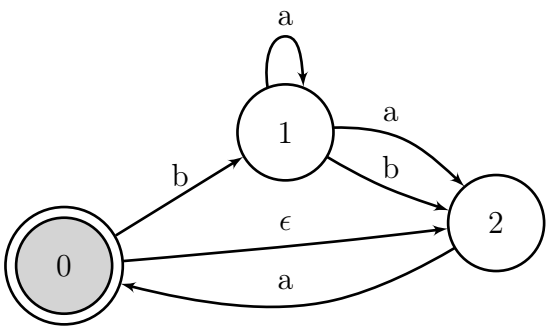
Дефиниция 2.2. Нека $\mathcal{A}$ е краен автомат. *Разширена релация на прехода* $\Delta^* \subseteq Q \times \Sigma^* \times Q$ дефинираме индуктивно:

- $\langle q, \epsilon, q \rangle \in \Delta^*$ 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 $w' = w_k \dots w_2 w_1 \in L(\mathcal{A}')$.

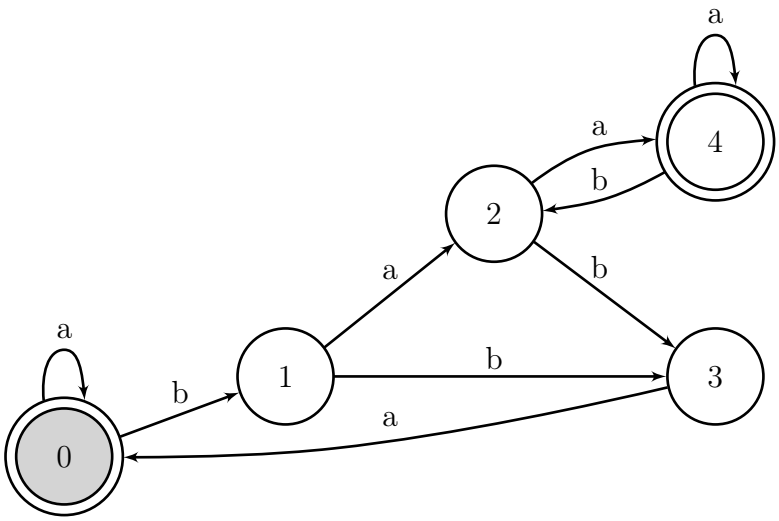
Дефиниция 2.7. Краен автомат $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta\rangle$ е *детерминиран*, ако:

- $\mathcal{A}$ 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

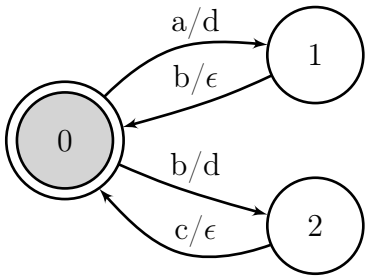
- Σ_1, Σ_2 са *крайни азбуки от символи*
- Q е *крайно множество от състояния*
- $I \subseteq Q$ е *множество от начални състояния*
- $F \subseteq Q$ е *множество от финални състояния*
- $\Delta \subseteq Q \times (\Sigma_1^* \times \Sigma_2^*) \times Q$ е *релация на прехода*

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 азбуки $\Sigma_1 = \{a, b, c\}, \Sigma_2 = \{d, \epsilon\}$ 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 *разлика* ($L_1 \backslash L_2$), *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 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*

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

С $\mathcal{O}_{\mathcal{B}} : \Sigma^* \to \Sigma^*$ бележим функцията, представена от бимашината и казваме, че тя превежда думата w в m, ако $\mathcal{O}_{\mathcal{B}}(w) = m$, където m е резултат от конкатенацията на всички $\psi(l_{i-1}, a_i, r_i)$ $(1 \leq i \leq k)$.

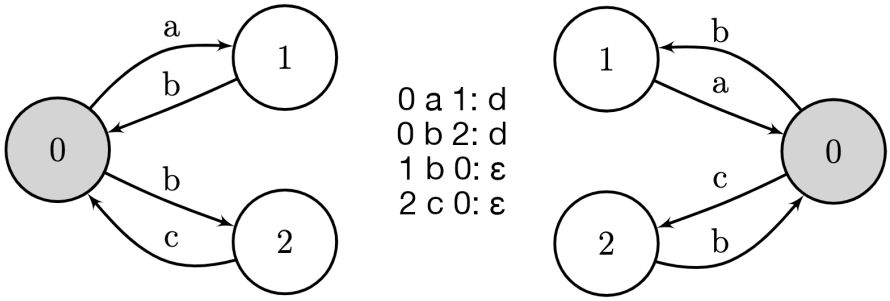
Бимашината чете входната дума и за всеки символ извежда дума над азбуката си. На всяка стъпка изведената от изходната функция ψ дума зависи от входния символ и двете състояния в които биха преминали левият и десният автомат, четейки входа съответно от ляво на дястно и от дясно на ляво. Крайният резултат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

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



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

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

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. [1] [6]

3 Правила за заместване

На двоичните стрингови релации можем да гледаме като на множество от преводи, като на пример за двойката думи $\langle u, v \rangle$, където $u, v \in \Sigma^*$, 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, 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 думата 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

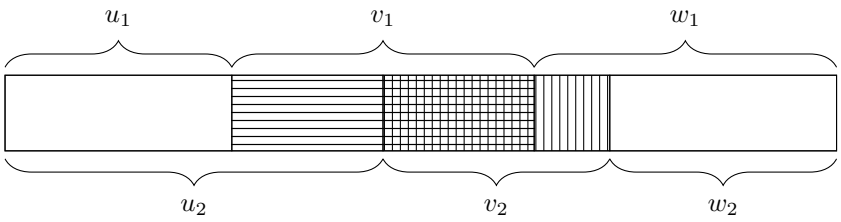
Правила за заместване можем да представим като *регулярни стрингови релации*, използвайки единствено алгебрата на регулярните езици и релации. Те се реализират програмно чрез крайни преобразуватели [2].

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

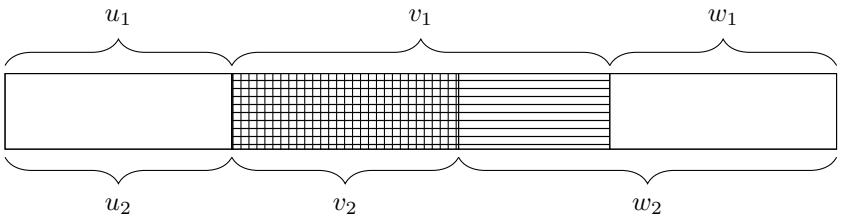
При прилагане на правило за заместване могат да възникнат многозначности в случаите, в които два контекста имат застъпващи се фокуси.



Фигура 3.2: Многозначност - контексти, чиито фокуси се застъпват

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 очвевидно 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 $\langle\epsilon, a, a\rangle$ и $\langle a, a, \epsilon\rangle$, и d с контекст $\langle\epsilon, aa, \epsilon\rangle$.

Дефиниция 3.4. Нека $A = \{\langle u_1, v_1, w_1\rangle, \ldots, \langle u_k, v_k, w_k\rangle\}$ е множество от *незастъпващи се* контексти на текст $t \in \Sigma^*$ и $k = |A|$, където $u_jv_j < u_{j+1}$ за $1 \leq i < j < k$. *Каноничното представяне* на t за A е редицата от $2k + 1$ думи $\langle u_1, v_1, x_2, v_2, \ldots, x_k, v_k, w_k\rangle$, където $x_i = [u_{i-1}v_{i-1}]^{-1}u_i$ за $2 \leq i \leq k$. Текстът t получаваме като конкатенираме на елементите в редицата $t = u_1v_1x_2v_2 \ldots x_kv_kw_k$. Очевидно е, че от тази редица също можем да изведем множеството от незастъпващи се контексти A.

Пример 3.4. Нека $t = aabcbcab, A = \{\langle a, ab, cbcab\rangle, \langle aabc, bc, ab\rangle, \langle aabcbc, ab, \epsilon\rangle\}$. *Каноничното представяне* на t за A е редицата

$$\langle u_1, v_1, x_2, v_2, x_3, v_3, w_3\rangle = \langle a, ab, c, bc, \epsilon, ab, \epsilon\rangle$$

Дефиниция 3.5. Въвеждаме следните оператори над множества от контексти:

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

$AFTER$ 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 3.6. Използвайки операторите от дефиниция 3.5, въвеждаме функцията $LML(A)$ (leftmost-longest). По зададено множество от контексти A, $LML(A)$ извежда множество от незастъпващи се контексти, като избира най-левите, най-дълги измежду тях.

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

където множествата C_i строим индуктивно

$$C_0 = \emptyset$$

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

$LML(A)$ е крайно множество, защото A е крайно, т.е. след даден 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 $\langle u_1, v_1, w_1\rangle$ и $\langle u_2, v_2, w_2\rangle C_{i_2}$ са били добавени първоначално в C_{i_1} и C_{i_2} съответно.

1. Нека $i_1 \geq i_2$, тогава

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

   След като $\langle u_2, v_2, w_2\rangle \notin C_{i_2-1}$, то

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

   От допускането, че $i_1 \geq i_2$ следва, че $\langle u_1, v_1, w_1\rangle, \langle u_2, v_2, w_2\rangle \in AFTER(A, C_{i_2-1})$ и $\langle u_2, v_2, w_2\rangle \in LEFTMOST(AFTER(A, C_{i_2-1}))$, което е в противоречие с дефиницията на $LEFTMOST$, тъй като $u_1 < u_2 < u_1v_1$.

2. Нека $i_1 < i_2$, тогава $\langle u_2, v_2, w_2\rangle \notin C_i$ 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 $AFTER$ (избира контекстите в A, чиито фокуси започват след края на тези в C_i, т.е. без да се застъпват) следователно $\langle u_2, v_2, w_2\rangle \notin C_{i+1}$. Това противоречи с факта, че $\langle u_2, v_2, w_2\rangle \in C_{i_2}$ и $i_1 < i_2$, от където заключваме, че $LML(A)$ не съдържа застъпващи се контексти.

□

Дефиниция 3.7. Нека $T \subseteq \Sigma^* \times \Sigma^*$ е регулярна релация и $t \in \Sigma^*$ е текст. С $A_{dom(T)}$ бележим множеството на всички поднизове в t, които са в $dom(T)$. Нека

$$t = u_1v_1x_2v_2 \dots x_kv_kw_k$$

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, ако

$$t' = u_1v'_1x_2v'_2 \dots x_kv'_kw_k$$

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. Това е очевидно поради факта, че думите u_1, x_i, w_k за $2 \leq i \leq k$ от каноничното представяне се превеждат с идентитет, а на всяка дума v_j, където $1 \leq j \leq k$ съответства точно една v'_j, така че $T(v_j) = v'_j$.

Пример 3.5. Нека разгледаме правилото `ab|bc` $\rightarrow d$, съответстващо на релацията $T = \{\langle ab, d\rangle, \langle bc, d\rangle\}$ и го приложим 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. Каноничното представяне на t е $\langle a, ab, cb, ab, \epsilon\rangle$ и резултатът от заместването е $R^{LML}(T)(aabcbab) = adcbd$.

3.2 Лексически анализ чрез регулярни релации

Регулярните релации намират своето приложение в множество домейни, като лексическият анализ е един от тях. Идеята е да сведем процеса до заместване на думи в текст, като по лексическата граматика построим релация на заместване под стратегията "най-ляво-най-дълго срещане". Тази релация реализираме програмно чрез краен преобразувател, който поставя маркери преди и след всяка разпозната лексема. В последствие по него строим еквивалентна бимашина следвайки [5], [6].

Ще представим формално конструкция на релация на заместване под стратегията най-ляво-най-дълго срещане, по алгоритъма на Карттунен (1996) [3]. Целта ни е по зададена регулярна релация T, да получим $R^{LML}(T)$. В последствие ще използваме тази конструкция, за да построим релация за лексически анализ.

Като начало дефинираме множеството на маркерите, които ще използваме по време на междинните стъпки за извършване на заместването - $\{cb, lb, rb\}$. Това са произволни символи извън входната азубука на T, $\Sigma \cap \{cb, lb, rb\} = \emptyset$. Множеството на всички символи бележим със $\Sigma_x = \Sigma \cup \{cb, lb, rb\}$.

Дефиниция 3.8. Представяме следните функции над регулярни езици:

$$not(L) = \{w \mid w \in \Sigma_x^* \wedge w \notin L\}$$

$$contain(L) = \{w \mid w_1 w_2 w_3 = w : w_1, w_3 \in \Sigma_x^* \wedge w_2 \in L\}$$

С $not(L)$ получаваме допълнението на езика L, $contain(L)$ връща езика, който се състои от всички думи, които съдържат инфикс в L. Строим функциите по следния начин:

$$not(L) := \Sigma_x^* \setminus L$$

$$contain(L) := \Sigma_x^* \cdot L \cdot \Sigma_x^*$$

Дефиниция 3.9. Представяме групата от оператори *intro*, въведени първоначално от Каплан и Кей (1994) [2]:

$$intro(S) := (Id(\Sigma_x \setminus S) \cup (\{\epsilon\} \times S))^*$$

$$introx(S) := (intro(S) \cdot Id(\Sigma_x \setminus S)) \cup \{\langle \epsilon, \epsilon \rangle\}$$

$$xintro(S) := (Id(\Sigma_x \setminus S) \cdot intro(S)) \cup \{\langle \epsilon, \epsilon \rangle\}$$

intro(S) създава релация, която добавя произволно символи от S в думи, които не съдържат символи от S, *introx(S)* и *xintro(S)* са аналогични, като символите от S не могат да се намират съответно в края и началото на изходната дума.

Пример 3.6. Имаме регулярният език $\{a, b\}$, в който *въвеждаме* символите от $S := \{x\}$. Тогава релацията $intro(S)$ ще съдържа двойките $\langle b, xb\rangle$, $\langle ab, axbx\rangle$, $\langle abba, axxbbax\rangle \ldots$ и т.н.

Дефиниция 3.10. Преставяме групата от *ignore* оператори както следва [2]:

$$ignore(L, S) := range(Id(L) \circ intro(S))$$

$$ignorex(L, S) := range(Id(L) \circ introx(S))$$

$$xignore(L, S) := range(Id(L) \circ xintro(S))$$

По даден регулярен език L и множество от символи S, *ignore(L,S)* връща регулярен език, който се състои от думите в L, с включени символи от S. *ignorex(L,S), xignore(L,S)* са аналогични като не включват символи от S съответно в края и в началото на думите.

Пример 3.7. Нека разгледаме регулярният език $L(E)$, представен от израза $E =$ `(a|b)+`. Този език се състои от всички думи с дължина поне един символ, които съдържат произволен брой *'a'* и *'b'*. $L(E) = \{a, b, ba, ab, ababb \ldots\}$, $ign(L(E), \{x\})$ ще съдържа думите *ab, xa, xab, xabxxb, bbb, bxabx ...*

Дефиниция 3.11. Въвеждаме следните функции, които строят филтриращи автомати по зададени регулярни езици [2]:

$$ifPthenS(P, S) = \{w \mid \forall w_1 w_2 = w : \text{ ако } w_1 \in P \text{ то } w_2 \in S\}$$

$$ifSthenP(P, S) = \{w \mid \forall w_1 w_2 = w : \text{ ако } w_2 \in S \text{ то } w_1 \in P\}$$

ifPthenS(P,S) дефинира езика, на чиито думи, ако префиксите им са в езика P то суфиксите са в S. Аналогично *ifSthenP(P,S)* представя думите, за които ако всичките им суфикси са в S, то префиксите им са в P. Реализираме тези функции както следва:

$$ifPthenS(P, S) := not(P \cdot not(S))$$

$$ifSthenP(P, S) := not(not(P) \cdot S)$$

Дефиниция 3.12. Комбинираме операторите от дефиниция 3.11, за да представим:

$$Piffs(P, S) = \{w \mid \forall w_1 w_2 = w : w_1 \in P \leftrightarrow w_2 \in S\}$$

$$Liffr(L, R) = \{w \mid \forall w_1 w_2 w_3 = w : \forall w_2' w_2'' = w_2 : w_2' \in L \leftrightarrow w_2'' \in R\}$$

PiffS(P,S) строи автомат, разпознаващ думите, чиито префикси са в P тогава и само тогава, когато суфиксите им са в S. *LiffR(L,R)* съдържа думите $w = w_1w_2w_3$, така че префиксите на w_2 са в L и суфиксите на w_2 са в R. Реализираме тези функции както следва:

$$PiffS(P,S) := ifPthenS(P,S) \cap ifSthenP(P,S)$$

$$LiffR(L,R) := PiffS((\Sigma_x^* \cdot L), (R \cdot \Sigma_x^*))$$

Дефиниция 3.13. Нека $T \subseteq \Sigma^* \times \Sigma^*$ е регулярна релация. *Релацията на задължително заместване* $R^{obl}(T)$ дефинираме както следва:

$$R^{obl}(T) := N(T) \cdot (T \cdot N(T))^*$$

$$N(T) = Id(\Sigma^* \setminus (\Sigma^* \cdot dom(T) \cdot \Sigma^*)) \cup \{\langle \epsilon, \epsilon \rangle\}$$

$N(T)$ е идентитета на множеството от всички думи, които не съдържат инфикси в $dom(T)$ обединен с двойката $\langle \epsilon, \epsilon \rangle$, заместваща празната дума с празната дума. Идеята е следната - поддумите от входа, които са в $dom(T)$ се превеждат спрямо T, докато тези, които не са в домейна ѝ се превеждат чрез идентитет, т.е. входът там остава непроменен.

Пример 3.8. Нека разгледаме следния пример: $T = \{\langle ab, d \rangle, \langle bc, d \rangle\}$ и имаме $R^{obl}(T)$. Входният текст $t_1 = babacbca$ се декомпозира като:

$$t_1 = b \cdot ab \cdot ac \cdot bc \cdot a$$

Сегментите $\{b, ac, a\} \in dom(N(T))$, докато $\{ab, bc\} \in dom(T)$. Съответно след заместването получаваме

$$t_1' = b \cdot d \cdot ac \cdot d \cdot a$$

За входният текст $t_2 = abcc$ обаче имаме две валидни композиции:

$$t_2 = ab \cdot cc = a \cdot bc \cdot c$$

Това ще доведе до два възможни превода, т.е. $\langle abcc, dcc \rangle \in R^{obl}(T)$ и $\langle abcc, adc \rangle \in R^{obl}(T)$.

Вече сме готови да дефинираме основните релации. Започваме с тази за първоначално срещане.

Дефиниция 3.14. *Релация на първоначално срещане* представяме както следва:

$$R^{init}(T) := intro(\{cb\}) \circ Id(LiffR(\{cb\}, xignore(dom(T), \{cb\})))$$

$R^{init}(T)$ поставя маркера `cb` в началото на всяко срещане на поддума (фокус) от входния текст, която е в домейна на T. Тази стъпка идентифицира началото на фокусите на всички контексти на заместване в текста независимо от това дали се застъпват, или са с възможно най-дълги фокуси.

Дефиниция 3.15. *Релация на най-ляво срещане* представяме както следва:

$$R^{left}(T) := (((Id(\Sigma^*) \cdot \{\langle cb, lb\rangle\} \cdot Id(ignorex(dom(T), \{\langle \epsilon, rb\rangle\})))^*) \cdot Id(\Sigma^*)) \circ R_{obl}(\Sigma_x, \{\langle cb, \epsilon\rangle\})$$

$R^{left}(T)$ получава текст с маркерите (`cb`) от първатата стъпка ($R^{init}(T)$) 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, като единият от тях маркира коректно контекстите спрямо стратегията "най-ляво-най-дълго срещане". Тази стъпка отговаря на операторите $AFTER$ и $LEFTMOST$ (дефиниция 3.5) и осигурява, че измежду контекстите със застъпващи се фокуси, ще останат тези, чийто фокус се намира най-ляво спрямо входния текст.

Дефиниция 3.16. *Релация на най-дълго срещане* представяме както следва:

$$R^{long}(T) := Id(not(contain(\{lb\} \cdot (ignorex(dom(T), \{lb, rb\}) \cap contain(\{rb\})))))$$

Релацията получава изхода от втората стъпка ($R^{left}(T)$) и измежду маркираните текстове съдържащи контексти с еднакво начало, извежда такъв, в който фокусите са най-дълги. Тази стъпка отговаря на оператора $LONGEST$.

Дефиниция 3.17. *Релация на заместване* представяме както следва:

$$R^{replace}(T) := R^{obl}(\Sigma_x, \{\langle lb, \epsilon\rangle\} \cdot T \cdot \{\langle rb, \epsilon\rangle\})$$

В тази стъпка получаваме изходния текст на R^{long}, 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 помощните маркери (*lb, rb*).

Пример 3.9. Имаме релациите R_1, R_2, съответстващи на правилата на заместване `ab+` $\rightarrow x$ и `bc` $\rightarrow y$. R_1 замества думите, започващи с a, последвано от едно, или повече b с x. R_2 замества думата bc с y. Релацията T получаваме като ги обединим: $T := R_1 \cup R_2$. Нека входният текст $t = abcabbdbc$ и маркерите `cb`, `lb` и `rb` представяме съответно чрез символите ^, < и >. Симулираме процеса на заместване както следва:

1. Маркираме началото на всички фокуси в думата:
   $R^{init}(T)(abcabbdbc) = \hat{}a\hat{}bc\hat{}abbd\hat{}bc$

2. Идентифицираме най-левите незастъпващи се контексти, което води до следните възможни декомпозиции:
   $\langle \hat{}a\hat{}bc\hat{}abbd\hat{}bc,\ <ab>c<ab>bd<bc> \rangle \in R^{left}(T)$
   $\langle \hat{}a\hat{}bc\hat{}abbd\hat{}bc,\ <ab>c<abb>d<bc> \rangle \in R^{left}(T)$

3. Релацията за най-дълго срещане филтрира думите, чиито маркери не следват съответното изискване:
   $<$ab$>$c$<$ab$>$bd$<$bc$> \notin dom(R^{long}(T))$
   $R^{long}(T)(<$ab$>$c$<$abb$>$d$<$bc$>) = <$ab$>$c$<$abb$>$d$<$bc$>$

4. Заместваме поддумите, оградени с маркери, спрямо релацията T и получаваме :
   $R^{long}(T)(<$ab$>$c$<$abb$>$d$<$bc$>) = xcxdy$

Дефиниция 3.18. *Релация на заместване под стратегията "най-ляво-най-дълго срещане"* получаваме като композираме релациите дефинирани в 3.14, 3.15, 3.16, 3.17 , които строим независимо една от друга.

$$R^{LML}(T) = R^{init}(T) \circ R^{left}(T) \circ R^{long}(T) \circ R^{replace}(T)$$

Дефиниция 3.19. Нека е дадена лексическата граматика

$$G := \langle E_1, E_2 \dots E_n \rangle$$

където $n = |G|$ и $E_i, i \in [1, n]$ са регулярни изрази, SoT, EoT са символи, с които съответно маркираме началото и края на лексемите във входния текст. *Релацията за лексически анализ* R^{lex} по зададената граматика G получаваме както следва:

$$T := \bigcup_{i=1}^{n} \{\langle \epsilon, i \cdot SoT \rangle\} \cdot Id(L(E_i)) \cdot \{\langle \epsilon, EoT \rangle\}$$

$$R^{lex} := R^{LML}(T)$$

Следствие 3.2. Ако T е функция, то R^{lex} е функция и по нея можем да построим еквивалентна бимашина. Това е изпълнено, ако не съществува дума $w \in \Sigma^*$ и регулярни изрази $E_i, E_j \in G$, така че $w \in L(E_i)$ и $w \in L(E_j)$ и $i \neq j$. Иначе казано, две правила в лексическата граматика не могат да разпознават една и съща дума.

Token	Description
E_1	`[0-9]+(\.[0-9]+)?`
E_2	`+|-|*|/`
E_3	`=`

Фигура 3.4: Лексическа граматика на аритметичен израз

Пример 3.10. Нека разгледаме лексическата граматика на Фигура 3.4, по която сме построили релация за лексически анализ R^{lex}. За входен текст $t = 99.9 + 5$, ще изведем следния резултат:

$$R^{lex}(99.9+5) = \texttt{1 SoT 99.9 EoT 2 SoT + EoT 1 SoT 5 EoT}$$

Изходният текст се състои от сегменти за всяка лексема, които са във вида *<type-index> SoT <token-text> EoT*.

3.3 Конструкция на бимашина

Крайните преобразуватели са еквивалентни по изразителност на регулярните релации. С тях също така можем да представим и регулярните фунцкии при условие, че преобразувателят е недетерминиран. Детерминираните преобразуватели от своя страна не са достатъчно изразителни, така че да покрият изцяло класа на регулярните функции. За разлика от тях, бимашините могат да моделират регулярните стрингови функции [1] като изцяло детерминистичен процес, с което задачите за заместване на думи в текст се решават за линейно време. В тази част ще представим формална конструкция на бимашина по зададен краен преобразувател, по алгоритъма представен в [5] и [6].

Нека е даден функционален, тримован, краен преобразувател $\mathcal{T} = \langle \Sigma \times \Sigma^*, Q, I, F, \Delta \rangle$, където $\langle \epsilon, \epsilon \rangle \in L(\mathcal{T})$. Целта ни е да построим бимашина $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$, така че $L(\mathcal{T}) = O_{\mathcal{B}}$.

Нека $\mathcal{A}_\mathcal{T} = \langle \Sigma, Q, I, F, \Delta_1 \rangle$ е подлежащият автомат на $\mathcal{T}$ и ${\mathcal{A}_\mathcal{T}}^{rev} = \langle \Sigma, Q, F, I, \Delta_1^{rev} \rangle$ е огледален на $\mathcal{A}_\mathcal{T}$. Десният автомат на бимашината, който симулираме по време на сканирането на входния текст от дясно на ляво, получаваме като детерминираме ${\mathcal{A}_\mathcal{T}}^{rev}$ и определим всички състояния като финални.

$$\mathcal{A}_\mathcal{R} = \mathcal{A}_{\mathcal{T}D}^{rev} = \langle \Sigma, Q_R, s_R, Q_R, \delta_R \rangle$$

където $Q_R \subseteq 2^Q$, $s_R = F$ и функцията на прехода дефинираме както следва:

$$\delta_R(R, a) = \{q \in Q \mid \exists q' \in R, m \in \Sigma^* : \langle q, \langle a, m \rangle, q' \rangle \in \Delta\}$$

За да възстановим успешните пътища в $\mathcal{T}$, ще обогатим състоянията на левият автомат с допълнителна информация. За целта дефинираме *функцията на избор на състояние* $\phi : Q_R \to Q$, която по дадено състояние $P \in Q_R$ на десния автомат (което представлява множество от състояния в $\mathcal{T}$) избира едно от тези състояния $\phi(P) = p$, където $p \in P \subseteq Q$. Дефинираме левия автомат на бимашината както следва:

$$\mathcal{A}_\mathcal{L} = \langle \Sigma, Q_L, s_L, Q_L, \delta_L \rangle$$

където множеството на състоянията $Q_L \subseteq 2^Q \times 2^{Q_R \times Q}$ са двойки, съставени от множества от състояния на $\mathcal{T}$ и функция за избор на състояние ϕ. Както при $\mathcal{A}_\mathcal{R}$, така и в $\mathcal{A}_L$ всички състояния са финални. Дефинираме състоянията и функцията на прехода както следва:

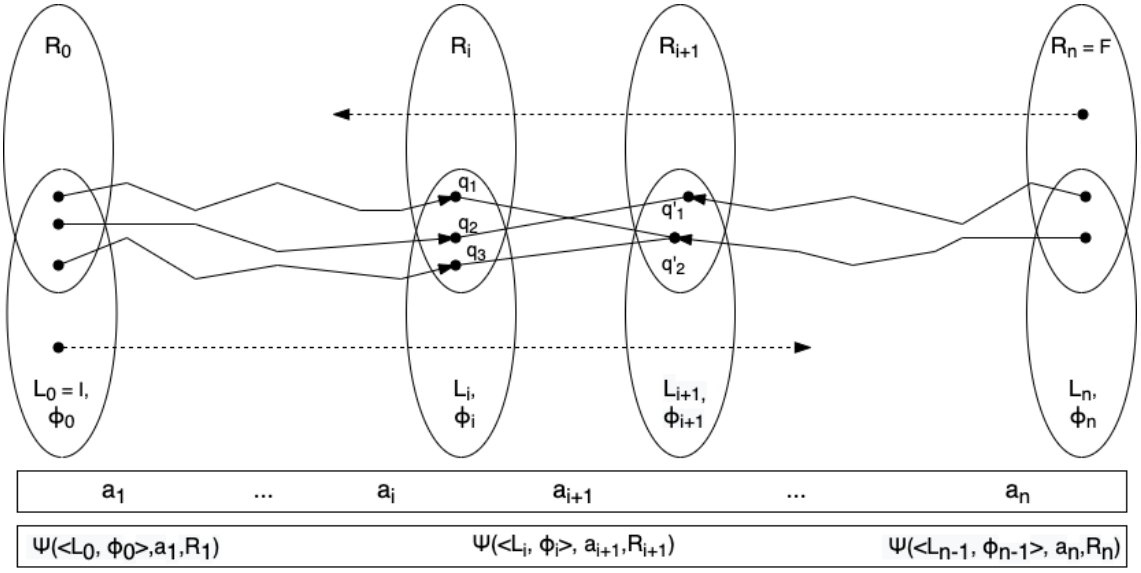
- $s_L := \langle I, \phi_0 \rangle$, $\phi_0(R) := \begin{cases} \text{произволен елемент от } R \cap I & \text{ако } R \cap I \neq \emptyset \\ \text{недефинирано} & \text{в прот. случай} \end{cases}$
- За всяко $\langle L, \phi \rangle \in Q_L$ и $a \in \Sigma$, $\delta_L(\langle L, \phi \rangle, a) := \langle L', \phi' \rangle$, където:
  - $L' := \{q' \mid \exists q \in L, m \in \Sigma^* : \langle q, \langle a, m \rangle, q' \rangle \in \Delta\}$
  - $\phi'(R') := \begin{cases} \text{произволен елемент от} \\ \{q' \in R' \mid \exists m \in \Sigma^* : \langle q, \langle a, m \rangle, q' \rangle \in \Delta\} & \text{ако } q = \phi(\delta_R(R', a)) \text{ е деф.} \\ \text{недефинирано} & \text{в прот. случай} \end{cases}$

От дефиницията на функцията на прехода δ_R следва, че ако $q = \phi(\delta_R(R', a))$ е дефинирано, то множеството $L' \neq \emptyset$ и $\phi'(R')$ може да се дефинира по посочения начин.

Дефинираме *функцията на изхода* ψ на бимашината, която по зададена двойка състояния $\langle L, \phi \rangle$ и R съответно на левия и десния автомат и буква $a \in \Sigma$, нека $\langle L', \phi' \rangle := \delta_L(\langle L, \phi \rangle, a)$ и $R := \delta_R(R', a)$. Тогава

$$\psi(\langle L, \phi \rangle, a, R') := \begin{cases} \text{произволен елемент от} \\ \{m \mid \langle \phi(R), \langle a, m \rangle, \phi'(R') \rangle \in \Delta\} & \text{ако } \phi(R) \text{ е деф.} \\ \text{недефинирано} & \text{в прот. случай} \end{cases}$$

Ако $q = \phi(R)$ е дефинирано, тогава $\{q' \in R' \mid \exists m \in \Sigma^* : \langle q, \langle a, m \rangle, q' \rangle \in \Delta\} \neq \emptyset$. От дефиницията на $\phi'(R')$ следва, че $\{m \mid \langle \phi(R), \langle a, m \rangle, \phi'(R') \rangle \in \Delta\} \neq \emptyset$



Фигура 3.5: Симулация на преобразувател в контекста на еквивалентната му бимашина.

и $\psi(\langle L, \phi\rangle, a, R')$ може да се дефинира по посочения начин. Детайлно доказателство на конструкцията е представено в оригиналната статия [5], както и в [6].

Нека разгледаме Фигура 3.5. Дадена е входна дума $w = a_1...a_n \in dom(\mathcal{T})$ и $\mathcal{T}$ се намира в множеството от начални състояния $I = L_0$. Когато $\mathcal{T}$ се намира в състояния $L_i, i \in [1, n]$ и прочете a_{i+1}, той преминава в L_{i+1}, следвайки всички преходи от L_i по символа a_{i+1}. Аналогично се извършва обхождането в обратната посока, където $F = R_n$ е множеството от начални състояния и R_0 са финалните и от състояня R_{i+1} на прочетен символ a_{i+1}, преминаваме в R_i следвайки преходите само че в обратна посока. Тъй като L_i и R_i се намират на успешни пътища съответно в левия и десния автомат на бимашината, то в съсоянията $\{q \mid q \in L_i \cap R_i\}$ се намират на успешен път в $\mathcal{T}$ и след прочитането на символа a_{i+1}, следвайки преходите от тези състояния, $\mathcal{T}$ преминава в състоянията $\{q' \mid q' \in L_{i+1} \cap R_{i+1}\}$. След като $\mathcal{T}$ е функционален, изходните думи

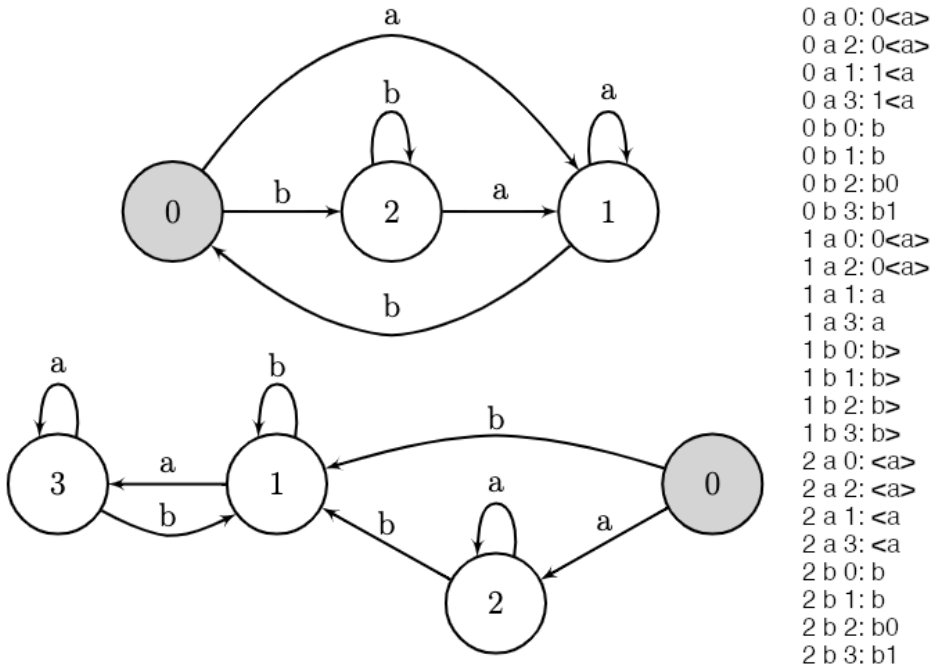
$$\{m \in \Sigma^* \mid \exists q \in L_i \cap R_i, \exists q' \in L_{i+1} \cap R_{i+1} : \langle q, \langle a_{i+1}, m\rangle, q'\rangle \in \Delta\}$$

на преходите от $L_i \cap R_i$ към $L_{i+1} \cap R_{i+1}$ съвпадат.

Пример 3.11. Нека разгледаме граматиката от Фигура 3.6 и построената по нея бимашина на Фигура 3.7 спрямо представената конструкция в тази глава.

Token	Description
E_1	`a`
E_2	`a+b`

Фигура 3.6: Лексическа граматика



Фигура 3.7: Бимашина за лексически анализ, построена по граматиката на Фигура 3.6

Представяме маркерите за начало и край на лексемата SoT, EoT съответно чрез символите $<$, $>$. Нека разгледаме входния текст *aabaa*. Симулираме левия и десния автомат на бимашината и получаваме следните пътища:

$$\pi_L = 0 \rightarrow^a 1 \rightarrow^a 1 \rightarrow^b 0 \rightarrow^a 1 \rightarrow^a 1$$

$$\pi_R = 3 \leftarrow^a 3 \leftarrow^a 1 \leftarrow^b 2 \leftarrow^a 2 \leftarrow^a 0$$

Резултата получаваме като конкатенираме резултатите от функцията на изхода

за всяка тройка, състояща се от ляво, дясно състояние и входна буква.

$$\begin{aligned}\mathcal{O}_{\mathcal{B}}(aabaa) &= \psi(0,a,3)\cdot\psi(1,a,1)\cdot\psi(1,b,2)\cdot\psi(0,a,2)\cdot\psi(1,a,0)\\ &= 1<a\cdot a\cdot b>\cdot 0<a>\cdot 0<a>=1<aab>0<a>0<a>\end{aligned} \tag{1}$$

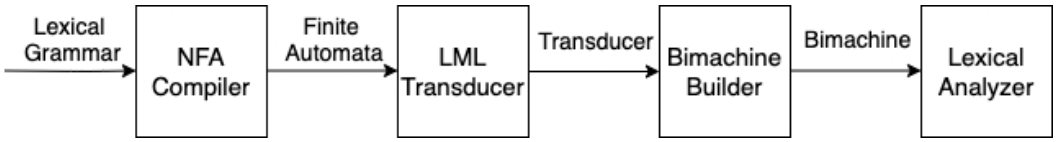
Всяка лексема е оградена от маркери, като преди маркера за начало имаме индекса на правилото в граматиката.

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 краен преобразувател по стратегията най-ляво-най-дълго срещане.
3. Конструкция на бимашина по зададен функционален преобразувател.
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 последната част ще видим как се осъществява лексически анализ чрез получената бимашина. Алгоритмите са имплементирани на езика C#.

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, демонстриращи представения 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:

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

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

⟨*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*⟩

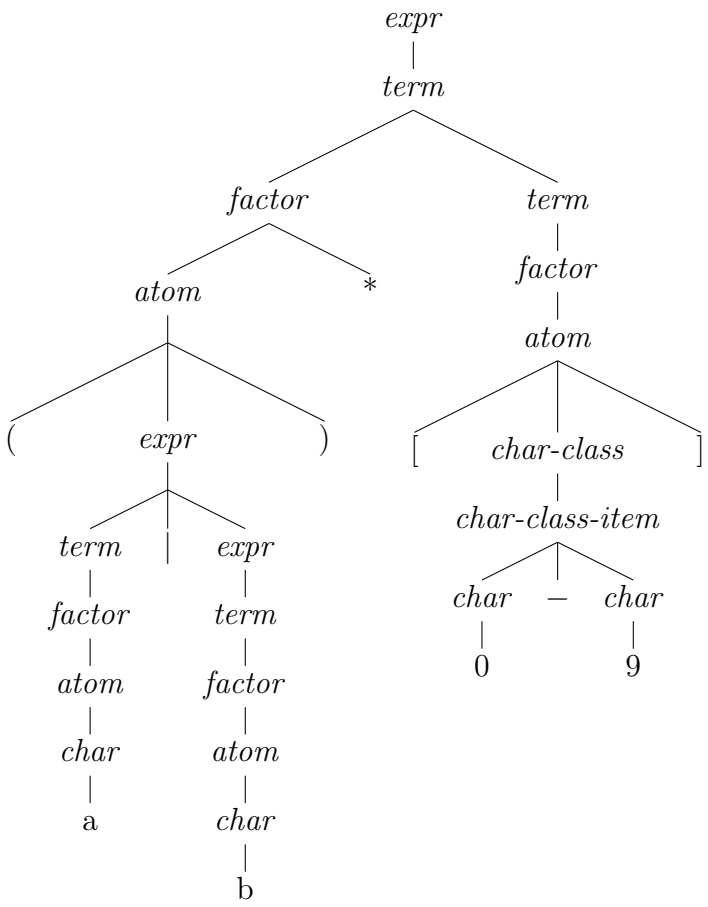
⟨*char*⟩ ::= *anyCharExceptMeta*
  | '\' ⟨*any-char*⟩

⟨*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 нула 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 [7]. 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 *парсър за 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 (в случая 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, така че по нея можем 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.

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

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
9            return FsaBuilder.FromSymbol(ch);
10     }
11     var ch = Next();
12     if (metaChars.Contains(ch))
```

```
13          throw new Exception($"Unescaped meta symbol {ch}");
14
15      return FsaBuilder.FromSymbol(ch);
16  }
```

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. Конвертираме регулярен израз към краен автомат по следния начин:

```
1       var fsa = new RegExp("(a|b)*c").Automaton;
```

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

Дефинираме метод, който по зададен краен преобразувател T и входна азбука, строи преобразувател по стратегията най-ляво-най-дълго срещане. Този метод извежда преобразувател, който е еквивалентен на релацията $R^{LML}(T)$. Алгоритъмът директно следва конструкцията, представена в секция 3.2. За целта сме имплементирали базовите алгоритми, реализиращи операциите над крайни автомати и преобразуватели (конкатенация, идентитет, разлика, композиция и пр.) представени в [4]. Реализираме фамилията от оператори *intro* и *ignore* по Каплан и Кей [2].

```
1   static Fst Intro(ISet<char> alphabet, ISet<char> symbols) =>
2       FsaBuilder.FromSymbolSet(alphabet.Except(symbols))
3           .Identity()
4           .Union(
5               FsaBuilder.FromEpsilon()
6                   .Product(FsaBuilder.FromSymbolSet(symbols)))
7           .Star();
8
9   static Fst IntroX(ISet<char> alphabet, ISet<char> symbols) =>
10      Intro(alphabet, symbols)
11          .Concat(FsaBuilder.FromSymbolSet(alphabet.Except(symbols)).Identity())
12          .Optional();
13
```

```
14  static Fst Xintro(ISet<char> alphabet, ISet<char> symbols) =>
15      FsaBuilder.FromSymbolSet(alphabet.Except(symbols))
16          .Identity()
17          .Concat(Intro(alphabet, symbols))
18          .Optional();
19
20  static Fsa Ignore(Fsa fsa, ISet<char> fsaAlphabet, ISet<char> symbols) =>
21      fsa.Identity()
22          .Compose(Intro(fsaAlphabet, symbols))
23          .Range();
24
25  static Fsa IgnoreX(Fsa fsa, ISet<char> fsaAlphabet, ISet<char> symbols) =>
26      fsa.Identity()
27          .Compose(IntroX(fsaAlphabet, symbols))
28          .Range();
29
30  static Fsa XIgnore(Fsa fsa, ISet<char> fsaAlphabet, ISet<char> symbols) =>
31      fsa.Identity()
32          .Compose(Xintro(fsaAlphabet, symbols))
33          .Range();
```

Релацията за задължително заместване $R^{obl}(T)$ в конктекста на краен преобразувател имплементираме както следва:

```
1   static Fst ToRewriter(this Fst fst, ISet<char> alphabet)
2   {
3       var all = FsaBuilder.FromSymbolSet(alphabet).Star();
4       var notInDomain = all
5           .Difference(all.Concat(fst.Domain()).Concat(all))
6           .Identity()
7           .Optional();
8
9       return notInDomain.Concat(fst.Concat(notInDomain).Star());
10  }
```

Вече можем да преминем към имплементацията на преобразувателя, представящ $R^{LML}(T)$.

```
1  public static Fst ToLmlRewriter(this Fst fst, ISet<char> alphabet)
2  {
3      if (alphabet.Intersect(markers).Any())
4          throw new ArgumentException("The alphabet contains invalid symbols.");
5
6      var alphabetStarFsa = FsaBuilder.FromSymbolSet(alphabet).Star().Minimal();
7      var allSymbols = alphabet.Concat(markers).ToHashSet();
8      var allSymbolsStarFsa = FsaBuilder.FromSymbolSet(allSymbols).Star().Minimal();
9
10     Fsa NotInLang(Fsa lang) => allSymbolsStarFsa.Difference(lang);
11     Fsa ContainsLang(Fsa lang) => allSymbolsStarFsa.Concat(lang, allSymbolsStarFsa);
12     Fsa IfPThenS(Fsa p, Fsa s) => NotInLang(p.Concat(NotInLang(s)));
13     Fsa IfSThenP(Fsa p, Fsa s) => NotInLang(NotInLang(p).Concat(s));
14     Fsa PiffS(Fsa l, Fsa r) => IfPThenS(l, r).Intersect(IfSThenP(l, r));
15     Fsa LiffR(Fsa l, Fsa r) => PiffS(allSymbolsStarFsa.Concat(l), r.Concat(allSymbolsStarFsa));
```

Първоначално инциализираме автомата, който разпознава думите, съставени от входната азбука (*alphabetStarFsa*) и този, който разпознава думите, съставени от входната азбука и маркерите включително (*allSymbolsStarFsa*) (6-9). В последствие дефинираме операторите над регулярни езици спрямо Каплан и Кей [2] (10-15).

```
16     var fstDomain = fst.Domain();
17     var initialMatch =
18         Intro(allSymbols, new HashSet<char> { cb })
19             .Compose(
20                 LiffR(
21                     FsaBuilder.FromSymbol(cb),
22                     XIgnore(fstDomain, allSymbols, new HashSet<char> { cb }))
23                 .Identity());
24
25     var leftToRight =
26         alphabetStarFsa.Identity()
27             .Concat(
28                 FstBuilder.FromWordPair(cb.ToString(), lb.ToString()),
29                 IgnoreX(fstDomain, allSymbols, new HashSet<char> { cb }).Identity(),
30                 FstBuilder.FromWordPair(string.Empty, rb.ToString()))
31             .Star()
```

```
32              .Concat(alphabetStarFsa.Identity())
33              .Compose(
34                  FstBuilder.FromWordPair(cb.ToString(), string.Empty).ToRewriter(allSymbols));
35
36      var includesNotLongestMatches =
37          ContainsLang(
38              FsaBuilder.FromSymbol(lb)
39                  .Concat(
40                      IgnoreX(fstDomain, allSymbols, new HashSet<char> { lb, rb })
41                          .Intersect(ContainsLang(FsaBuilder.FromSymbol(rb)))));
42      var longestMatch = NotInLang(includesNotLongestMatches).Identity();
43
44      var replacement =
45          FstBuilder.FromWordPair(lb.ToString(), string.Empty)
46              .Concat(
47                  fst,
48                  FstBuilder.FromWordPair(rb.ToString(), string.Empty))
49              .ToRewriter(allSymbols);
50
51      return initialMatch.Compose(leftToRight, longestMatch, replacement);
52  }
```

Строим преобразувателите, съответстващи на релациите $R^{init}(T)$ (17-23), $R^{left}(T)$ (25-34), $R^{long}(T)$ (36-42), $R^{replace}(T)$ (44-49). Преобразувателят под стратегията най-ляво-най-дълго срещане $R^{LML}(T)$ получаваме като композираме четири преобразувателя в съответния ред (51).

4.4 От краен преобразувател към бимашина

Ако, преобразувателят, построен по стратегията най-ляво-най-дълго срещане е функционален, по него можем да построим еквивалентна в бимашина. Реализираме алгоритъма представен в [5] и [6].

Първата стъпка е да детерминираме подлежащия огледален автомат на входния преобразувател.

```
1   public Bimachine ToBimachine(this Fst rtFst, ISet<char> alphabet)
2   {
3       var fstTransGroupedByTarget = rtFst.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>> { rtFst.Final.ToHashSet() };
10      var rightDfaTrans = 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              rightDfaTrans.Add(
25                  (n, label.Single()),
26                  rightSStates.FindIndex(ss => ss.SetEquals(targetSState)));
27      }
```

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.

```
28      var rDfaTransGroupedByTarget = rightDfaTrans
29          .GroupBy(kvp => kvp.Value, kvp => (To: kvp.Key.From, Symbol: kvp.Key.Label))
30          .ToDictionary(g => g.Key, g => g);
31
32      var fstTransGroupedBySourceAndSymbol =
33          new Dictionary<(char In, int From), ISet<(int To, string Out)>>();
34
```

```
35      foreach (var tr in rtFst.Transitions) {
36          if (!fstTransGroupedBySourceAndSymbol.ContainsKey((tr.In.Single(), tr.From)))
37              fstTransGroupedBySourceAndSymbol[(tr.In.Single(), tr.From)] =
38                  new HashSet<(int, string)>();
39
40          fstTransGroupedBySourceAndSymbol[(tr.In.Single(), tr.From)].Add((tr.To, tr.Out));
41      }
42
43      var leftDfaStates = new List<(ISet<int> SState, IDictionary<int, int> Selector)>();
44      var leftDfaTrans = new Dictionary<(int, char), int>();
45      var bmOutput = new Dictionary<(int, char, int), string>();
```

Групираме преходите на десния автомат по тяхната дестинация за оптимален досъп (28-30). Аналогично групираме преходите на входния преобразувател по входен символ и състояние (32-41). Инициализираме списък, в който ще съхраняваме състоянията, преходите на левия автомат и изходната функция на бимашината (43-45).

```
46      var initStateSelector = new Dictionary<int, int>();
47
48      for (int rIndex = 0; rIndex < rightSStates.Count; rIndex++) {
49          var initStates = rightSStates[rIndex].Intersect(rtFst.Initial);
50          if (initStates.Any())
51              initStateSelector.Add(rIndex, initStates.First());
52      }
53
54      leftDfaStates.Add((rtFst.Initial.ToHashSet(), initStateSelector));
```

Инициализираме L_0, ϕ_0, следвайки директно формалната дефиниция (46-54). Следва да реализираме индукцията, по която строим състоянията и преходите на левия автомат и изходната функция на бимашината.

```
55      for (int k = 0; k < leftDfaStates.Count; k++) {
56          var currLState = leftDfaStates[k];
57          var targetLStatesPerSymbol =
58              new Dictionary<char, (ISet<int> LSState, IDictionary<int, int> Selector)>();
59
```

```
60          foreach (var symbol in alphabet) {
61              var targetLSState = new HashSet<int>();
62
63              foreach (var st in currLState.SState)
64                  if (fstTransGroupedBySourceAndSymbol.ContainsKey((symbol, st)))
65                      targetLSState.UnionWith(
66                          fstTransGroupedBySourceAndSymbol[(symbol, st)].Select(x => x.To));
67
68              if (!targetLSState.Any())
69                  continue;
70
71              var targetSelector = new Dictionary<int, int>();
72
73              foreach (var (toRIndex, fstState) in currLState.Selector) {
74                  if (!rDfaTransGroupedByTarget.ContainsKey(toRIndex))
75                      continue;
76
77                  foreach (var (fromRIndex, _) in
78                      rDfaTransGroupedByTarget[toRIndex].Where(p => p.Symbol == symbol)) {
79
80                      if (!fstTransGroupedBySourceAndSymbol.ContainsKey((symbol, fstState)))
81                          continue;
82
83                      var reachableFstState = fstTransGroupedBySourceAndSymbol[(symbol, fstState)]
84                          .FirstOrDefault(p => rightSStates[fromRIndex].Contains(p.To));
85
86                      if (reachableFstState != default)
87                          targetSelector.Add(fromRIndex, reachableFstState.To);
88                  }
89              }
90
91              if (targetSelector.Any())
92                  targetLStatesPerSymbol.Add(symbol, (targetLSState, targetSelector));
93          }
94
95          foreach (var (symbol, (targetLSState, targetSelector)) in targetLStatesPerSymbol) {
96              foreach (var (fromRIndex, fstState) in targetSelector) {
97                  var predecessorOfR = rightDfaTrans[(fromRIndex, symbol)];
98                  var state = currLState.Selector[predecessorOfR];
99                  var destinations = fstTransGroupedBySourceAndSymbol[(symbol, state)]
```

```
100                     .Where(p => p.To == fstState);
101
102                 foreach (var (toState, word) in destinations) {
103                     var outFnPair = (Key: (k, symbol, fromRIndex), Val: word);
104
105                     if (bmOutput.ContainsKey(outFnPair.Key)) {
106                         if (bmOutput[outFnPair.Key] != outFnPair.Val)
107                             throw new InvalidOperationException(
108                                 $"Cannot have different values for the same key:"
109                                     + "'{bmOutput[outFnPair.Key]}', '{outFnPair.Val}'");
110                     }
111                     else bmOutput.Add(outFnPair.Key, outFnPair.Val);
112                 }
113             }
114
115             var nextLState = (targetLSState, targetSelector);
116
117             if (!leftDfaStates.Any(ls => AreBmLeftStatesEqual(ls, nextLState)))
118                 leftDfaStates.Add(nextLState);
119
120             leftDfaTrans.Add(
121                 (k, symbol),
122                 leftDfaStates.FindIndex(ls => AreBmLeftStatesEqual(ls, nextLState)));
123         }
124     }
```

Пояснения ...

```
125     var leftStateIndices = Enumerable.Range(0, leftDfaStates.Count);
126     var leftDfa = new Dfsa(leftStateIndices, 0, Array.Empty<int>(), leftDfaTrans);
127     var rightStateIndices = Enumerable.Range(0, rightSStates.Count);
128     var rightDfa = new Dfsa(rightStateIndices, 0, Array.Empty<int>(), rightDfaTrans);
129
130     return new Bimachine(leftDfa, rightDfa, bmOutput);
131 }
```

Завършваме конструкцията с представянето на автоматите на бимашината (125-131). Състоянията им бележим с индексите на елементите в съответните множества. Тъй като всяко състояние в тях е финално, не се налага да съхраняваме изрично множество от финални състояния.

4.5 Лексически анализ чрез бимашина

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

```
var lexer = Lexer.Create(new[] {
    new Rule("[0-9]+\\.?[0-9]*", "NUM"),
    new Rule("[+*/-]", "OP"),
    new Rule("=", "EQ"),
});
lexer.Input = new InputStream("3.14+1.86=5");

foreach (var token in lexer.GetNextToken())
    Console.WriteLine(token);
```

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

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

Разглеждаме процедурата по създаване на лексическия анализатор, следвайки Дефиниция 3.19

```
static Lexer Create(IList<Rule> grammar)
{
    var tokenFsts = new List<Fst>();

    for (int i = 0; i < grammar.Count; i++) {
        var ruleFsa = new RegExp(grammar[i].Pattern).Automaton;
        var ruleFst = FstBuilder.FromWordPair("", $"{i}{SoT}")
            .Concat(ruleFsa.Identity())
            .Concat(FstBuilder.FromWordPair("", $"{EoT}"));
```

```
10          tokenFsts.Add(ruleFst);
11      }
12
13      var unionFst = tokenFsts.Aggregate((u, f) => u.Union(f));
14      var alphabet = unionFst.InputAlphabet
15          .Where(s => !string.IsNullOrEmpty(s))
16          .Select(s => s.Single())
17          .ToHashSet();
18
19      var lmlFst = unionFst.ToLmlRewriter(alphabet);
20      var bm = lmlFst.ToBimachine(alphabet);
21
22      return new Lexer(bm, grammar);
23  }
```

На редове (0-0) декларираме символите, с които ще означаваме началото и края на лексемите. На (0-0) 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 (0-0) обединяваме тези преобразуватели и определяме азбуката като вземем уникалните символи по горната лента на получения преобразувател. От него на ред (0) строим такъв под стратегията "най-ляво-най-дълго"срещане, който превръщаме в еквивалентна бимашина (0).

```
1   public class Lexer
2   {
3       const char SoT = '\u0002';
4       const char EoT = '\u0003';
5       readonly 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; private set; }
14      public InputStream Input { get; set; }
```

Лексическият анализатор съдържа списък от правила (лексическата граматика) и бимашина, построена по съответната граматика. Типът *InputStream* е интерфейс 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.Right.ReverseRecognitionPath(Input);
18
19          if (rPath.Count != Input.Size + 1)
20              throw new ArgumentException(
21                  $"Invalid input symbol. {Input.CharAt(Input.Size - rPath.Count)}");
22
23          var leftState = Bm.Left.Initial;
24          var token = new StringBuilder();
25          var typeIndex = new StringBuilder();
26          var tokenIndex = 0;
27          var tokenStartPos = 0;
28
29          for (Input.SetToStart(); !Input.IsExhausted; Input.MoveForward()) {
30              var ch = Input.Peek();
31              var rightIndex = rPath.Count - 2 - Input.Pos;
32              var triple = (leftState, ch, rPath[rightIndex]);
33
34              if (!Bm.Output.ContainsKey(triple))
35                  throw new ArgumentException($"Invalid token '{token.ToString() + ch}'");
36
37              token.Append(Bm.Output[triple]);
38
39              if (token[token.Length - 1] == EoT)
40              {
41                  token.Remove(token.Length - 1, 1);
42                  for (var i = 0; token[i] != SoT; i++)
43                      typeIndex.Append(token[i]);
44                  token.Remove(0, typeIndex.Length + 1);
45
46                  yield return new Token
```

```
47                  {
48                      Index = tokenIndex,
49                      Position = (tokenStartPos, Input.Pos),
50                      Text = token.ToString(),
51                      Type = grammar[int.Parse(typeIndex.ToString())].Name
52                  };
53
54                  token.Clear();
55                  typeIndex.Clear();
56                  tokenIndex++;
57                  tokenStartPos = Input.Pos + 1;
58              }
59
60              if (!Bm.Left.Transitions.ContainsKey((leftState, ch)))
61                  throw new ArgumentException($"Invalid input. {ch}");
62
63              leftState = Bm.Left.Transitions[(leftState, ch)];
64          }
65      }
66  }
```

На ред (0) извличаме редицата от състояния получена от симулацията на десния авомат, прочитайки входния текст от дясно на ляво. Ако автоматът не е прочел целия текст, то входът е невалиден връщаме грешка (0-0). Започваме симулацията на левия автомат, заедно с извеждането на лексемите. Намираме се в началното му състояние (0), променливата *token* (0) използваме, за да съхраняваме текста на прочетената лексема, *typeIndex* (0) съдържа индекса на правилото в граматиката, чрез който определяме какъв тип е изведената лексема, *tokenIndex* (0) определя коя по ред лексема сме извели, а *tokenStartPos* (0) 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 (0-0). 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 (0-0). Ако последният символ на изведената дума е маркер за край на лемсема (end of token) (0), 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 ели-

миниране сегмента, така че в *token* да остане само текста на лексемата (0-0), докато *tokenIndex* съдържа идекса на правилото в граматиката. На редове (0-0) 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 (0-0), което означава, че текстът съдържа невалидна лексема.

Литература

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

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

[3] Karttunen, Lauri. (1996) *Directed Replacement.* In Joshi, A. and Palmer, M., editors, Proceedings of the 34th Annual Meeting of the As- sociation for Computational Linguistics, pages 108–115, Santa Cruz.

[4] S., Mihov, S., and Schulz, K. U. (2018) *Finite-State Techniques Automata, Transducers and Bimachines* Chapter 11 "Constructing finite-state devices for text rewriting"

[5] 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.

[6] S., Mihov, S., and Schulz, K. U. (2018) *Finite-State Techniques Automata, Transducers and Bimachines* Chapter 6.2 "Equivalence of regular string funcitons and bimachines."

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