МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра: Математической физики и оптимального управления

Направление подготовки: «Математика и компьютерные науки»

КУРСОВАЯ РАБОТА

Тема:
«Лексический анализатор»

Выполнил(а): студент группы 381505
Чикмарев Илья Валерьевич
_____ И. В. Чикмарев
«____» _____ 2017 г.

Научный руководитель: кандидат физико-математических наук, доцент
Гаврилов Владимир Сергеевич
_____ В. С. Гаврилов
«____» _____ 2017 г.

Нижний Новгород
2017

Оглавление

Глава 1. Сведения из теории формальных языков

Данная глава посвящена минимально необходимым для реализации лексического анализа сведениям из теории формальных языков и конечных автоматов.

1.1. Определение алфавитов и языков

Прежде всего приведём определение алфавита и языка.

Алфавитом называется любое конечное множество некоторых символов. При этом понятие символа не определяется, поскольку оно в теории формальных языков является базовым.

Как правило, алфавит будем обозначать заглавными греческими буквами (например, буквой Σ), возможно, с нижними индексами.

Приведём примеры алфавитов:

1) $\{0, 1\}$ — алфавит Σ_1, состоящий из нуля и единицы;
2) $\{A, B, \ldots, Z\}$ — алфавит Σ_2, состоящий из заглавных латинских букв;
3) $\{\text{А}, \text{Б}, \text{В}, \Gamma, \text{Д}, \text{Е}, \text{Ё}, \ldots, \text{Я}\}$ — алфавит Σ_3, состоящий из заглавных русских букв;
4) {**int**, **void**, **return**, *, (,), '{', '}', ;, main, *number*} — алфавит Σ_4, состоящий из ключевых слов **int**, **void**, **return** языка Си, идентификатора main, звёздочки, круглых скобок, фигурных скобок, точки с запятой, и целых чисел *number* (синтаксис целых чисел — как в языке Си);
5) $\{a_1, a_2, a_3, a_4\}$ — алфавит Σ_5, состоящий из каких–то четырёх символов.

Из символов алфавита можно составлять **строки**, то есть конечные последовательности символов. Если строка состоит из символов алфавита Σ, то её называют **строкой над алфавитом** Σ. **Длиной строки** x называется количество символов в этой строке. Длину строки x будем обозначать $|x|$. Строка, вообще не содержащая символов, называется **пустой строкой** и будет обозначаться ε.

Приведём примеры строк:

1) 0111001 — строка над алфавитом Σ_1;
2) ENGLISH, INTEL — строки над алфавитом Σ_2;
3) МОСКВА, ГОРЬКИЙ, АЛЁШКОВО — строки над алфавитом Σ_3;
4) **int** main (**void**)'{' **return** *number*';' '}' — строка над алфавитом Σ_4;
5) $a_1a_3a_2a_2a_4$ — строка над алфавитом Σ_5.

Далее потребуется операция **сцепления** (иногда говорят **конкатенации**) строк. Эта операция заключается в приписывании одной строки в конец другой. Например, если строки α и β таковы, что $\alpha = abc$, $\beta = defg$, то сцепление строк α и β обозначается $\alpha\beta$, и представляет собой строку $abcdefg$.

Множество всех строк над алфавитом Σ обозначается Σ^*. Скажем, если $\Sigma = \{0, 1\}$, то $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots, 1010, \ldots\}$. Ясно, что множество всех строк над заданным алфавитом — бесконечно.

Любой набор строк над некоторым алфавитом называется **языком** (ещё называют **формальным языком**, чтобы отличать от естественных языков). Допустим, из всевозможных строк над алфавитом $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '.', -\}$ можно выбрать те, которые являются корректной записью некоторого вещественного числа: $L = \{0, -1.5, 1002.123345, 777, \ldots\}$. Языки обозначаются заглавными латинскими буквами, возможно с нижними индексами. Язык может являться конечным множеством строк: если L — язык над алфавитом $\{a, b\}$, содержащий лишь строки состоящие из менее чем трёх символов, то $L = \{\varepsilon, a, b, aa, ab, ba, bb\}$.

1.2. Операции над языками и регулярные выражения

Поскольку язык — это некоторое множество строк, то нужно уметь это множество как–то описывать. Одним из способов описания являются так называемые регулярные выражения. Прежде чем определить, что такое регулярное выражение, нужно определить операции над языками. Операции над языками, которые нам потребуются, собраны в табл.1.2.1.

Таблица 1.2.1. Операции над языками.

Операция	Определение и обозначение операции
Объединение L и M	$L \cup M = \{s : s \in L \text{ или } s \in M\}$
Сцепление L и M	$LM = \{st : s \in L \text{ и } t \in M\}$
Замыкание Клини́ языка L	$L^* = \bigcup\limits_{i=0}^{\infty} L^i$
Положительное замыкание языка L	$L^+ = \bigcup\limits_{i=1}^{\infty} L^i$

В этой таблице L и M — некоторые языки над алфавитом Σ. Кроме того, в таблице используется обозначение L^i, которое означает следующее: $L^0 = \{\varepsilon\}$, $L^i = \underbrace{L \dots L}_{i \text{ раз}}$.

Определение 1.2.1. *Регулярные выражения строятся из подвыражений, в соответствии с описанными ниже правилами. В этих правилах через $L(r)$ обозначен язык, описываемый регулярным выражением r. Правила построения регулярных выражений таковы:*

1) *ε — регулярное выражение, и $L(\varepsilon) = \{\varepsilon\}$;*

2) *если a — символ алфавита Σ, то a — регулярное выражение, и $L(a) = \{a\}$;*

3) *если r и s — регулярные выражения, то $(r)|(s)$ — тоже регулярное выражение, и $L((r)|(s)) = L(r) \cup L(s)$;*

4) *если r и s — регулярные выражения, то и $(r)(s)$ — регулярное выражение, причём $L((r)(s)) = L(r)L(s)$;*

5) *если r — регулярное выражение, то $(r)^*$ также является регулярным выражением, и $L((r)^*) = (L(r))^*$;*

6) *если r — регулярное выражение, то и (r) — регулярное выражение, причём $L((r)) = L(r)$;*

7) *ничто иное не является регулярным выражением.*

Записанные в соответствии с этим определением регулярные выражения часто содержат лишние пары скобок. Многие скобки можно опустить, если принять следующие соглашения:

1) унарный оператор * — левоассоциативен (т.е. выполняется слева направо) и имеет наивысший приоритет;

2) сцепление имеет второй по величине приоритет и также левоассоциативно;

3) оператор $|$ — левоассоциативен, и имеет наименьший приоритет.

Пользуясь данными соглашениями, регулярное выражение $(a)|((b)^*c)$ можно переписать в виде $a|b^*c$.

Приведём примеры регулярных выражений.

Пример 1.2.1. *Пусть $\Sigma = \{a, b\}$. Тогда*

1) *регулярное выражение $a|b$ описывает язык $\{a, b\}$;*

2) *регулярное выражение $(a|b)(a|b)$ описывает язык $\{aa, ab, ba, bb\}$ над алфавитом Σ; другое регулярное выражение для того же языка — $aa|ab|ba|bb$;*

3) *регулярное выражение a^* описывает язык, состоящий из всех строк из нуля или более символов a, т.е. язык $\{\varepsilon, a, aa, aaa, \dots\}$;*

4) *регулярное выражение* $(a|b)^*$ *описывает множество всех строк из символов* a *и* b*:* $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$*; другое регулярное выражение для того же языка:* $(a^*|b^*)^*$*;*

5) *регулярное выражение* $a|a^*b$ *описывает язык* $\{a, b, ab, aab, aaab, \dots\}$.

Язык, который может быть определён регулярным выражением, называется **регулярным языком**. Если два регулярных выражения, r и s, описывают один и тот же язык, то выражения r и s называются **эквивалентными**, что записывается как $r = s$. Для регулярных выражений есть ряд алгебраических законов, каждый из которых заключается в утверждении об эквивалентности двух разных регулярных выражений. В табл.1.2.2 приведены некоторые такие законы. Другие алгебраические законы для регулярных выражений можно найти, например, в [**?**].

Таблица 1.2.2. Некоторые алгебраические законы для регулярных выражений.

Закон	Описание
$r\|s = s\|r$	Оператор $\|$ — коммутативен.
$r\|(s\|t) = (r\|s)\|t$	Оператор $\|$ — ассоциативен.
$r(st) = (rs)t$	Сцепление — ассоциативно.
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	Сцепление дистрибутивно относительно оператора $\|$.
$\varepsilon r = r\varepsilon = r$	ε является нейтральным элементом по отношению к сцеплению строк.
$r^* = (r\|\varepsilon)^*$	ε гарантированно входит в замыкание Клини́.
$(r^*)^* = r^*$	Оператор * — идемпотентен.

Для удобства записи регулярным выражениям можно присваивать имена и использовать затем эти имена в последующих выражениях так, как если бы это были символы. Если Σ — некоторый алфавит, то **регулярным определением** называется последовательность определений вида

$$\begin{aligned} d_1 &\to r_1 \\ d_2 &\to r_2 \\ &\dots \\ d_n &\to r_n \end{aligned}$$

Здесь

1) каждое d_i — новый символ, не входящий в Σ и не совпадающий ни с каким иным d_i;

2) каждое r_i при $i > 1$ — регулярное выражение над алфавитом $\Sigma \cup \{d_1, \dots, d_{i-1}\}$, а r_1 — регулярное выражение над алфавитом Σ.

Пример 1.2.2. *Идентификаторы языка Си представляют собой строки из латинских букв, десятичных цифр, и знаков подчёркивания, причём идентификатор не должен начинаться с десятичной цифры. С помощью регулярных определений это можно записать так:*

$$\begin{aligned} &\textit{буква} \to A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z| \\ &\qquad a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z \\ &\textit{буква\_или\_подчерк} \to \textit{буква}|\_ \\ &\textit{цифра} \to 0|1|2|3|4|5|6|7|8|9 \\ &\textit{идентификатор} \to \textit{буква\_или\_подчерк}(\textit{буква\_или\_подчерк}|\textit{цифра})^* \end{aligned}$$

Пример 1.2.3. *Пусть беззнаковые числа (целые и с плавающей запятой) представляют собой строки вида* 5280*;* 0.01234*;* $6.336E4$*;* $1.89E-4$*. Точную спецификацию этого множества строк можно записать в виде следующего регулярного определения:*

$$\begin{aligned} &\textit{цифра} \to 0|1|2|3|4|5|6|7|8|9 \\ &\textit{цифры} \to \textit{цифра цифра}^* \\ &\textit{необяз\_дробная\_часть} \to .\textit{цифры}|\varepsilon \\ &\textit{необяз\_экспонента} \to E(+|-|\varepsilon)\textit{цифры}|\varepsilon \\ &\textit{число} \to \textit{цифры необяз\_дробная\_часть необяз\_экспонента} \end{aligned}$$

С тех пор, как в 1950–х Клини́ ввёл регулярные выражения с базовыми операторами объединения, сцепления, и замыкания Клини́, к регулярным выражениям добавлено много расширений, о которых можно прочитать, скажем, в [**?**]. Упомянем лишь некоторые из них.

1) *Один или несколько экземпляров.* Унарный постфиксный оператор $^+$ представляет положительное замыкание регулярного выражения и его языка. Иначе говоря, если r — регулярное выражение, то $(r)^+$ описывает язык $(L(r))^+$. Опрератор $^+$ имеет те же приоритет и ассоциативность, что и оператор *. Замыкание Клини́ и положительное замыкание связывают два алгебраических закона: $r^* = r^+|\varepsilon$ и $r^+ = rr^* = r^*r$.

2) *Нуль или один экземпляр.* Унарный постфиксный оператор ? означает „нуль или один экземпляр“, то есть запись $r?$ представляет собой сокращение для $r|\varepsilon$. Иными словами, $L(r?) = L(r) \cup \{\varepsilon\}$. Оператор ? имеет те же приоритет и ассоциативность, что и операторы * и $^+$.

3) *Классы символов.* Регулярное выражение $a_1|\ldots|a_n$, где a_i, $i = \overline{1,n}$, — символы алфавита, можно переписать сокращённо: $[a_1 \ldots a_n]$. При этом если символы $a_1, \ldots, a_n$ образуют логическую последовательность (например, последовательные прописные буквы, последовательные строчные буквы, десятичные цифры), то выражение $a_1|\ldots|a_n$ можно заменить выражением $[a_1 - a_n]$. Например, $a|b|c$ можно переписать в виде $[abc]$, а $a|\ldots|z$ — в виде $[a \ldots z]$ или $[a - z]$.

Приведём примеры использования этих трёх расширений.

Пример 1.2.4. *С помощью указанных расширений определение идентификаторов языка Си можно записать так:*

$$\textit{буква} \to [A - Za - z]$$
$$\textit{буква\_или\_подчерк} \to \textit{буква}|\_$$
$$\textit{цифра} \to [0 - 9]$$
$$\textit{идентификатор} \to \textit{буква\_или\_подчерк}(\textit{буква\_или\_подчерк}|\textit{цифра})^*$$

Определение же беззнаковых чисел можно переписать так:

$$\textit{цифра} \to [0 - 9]$$
$$\textit{цифры} \to \textit{цифра}^+$$
$$\textit{число} \to \textit{цифры}(.\textit{цифры})?(E[+-]?\textit{цифры})?$$

1.3. Конечные автоматы

На данный момент мы знаем, что регулярный язык — это язык, определяемый регулярным выражением. Однако встаёт вопрос: как определять, принадлежит ли строка регулярному языку или нет (или, как ещё говорят, как распознавать регулярные языки)?

Запрограммировать распознавание позволяют **конечные автоматы**. Прежде всего скажем о **детерминированных конечных автоматах**.

Автомат — это некоторое устройство с конечным числом состояний. Среди всех состояний можно выделить „особые“: это **начальное** состояние, и одно или более **допускающих** (или **конечных**) состояний. Начальное состояние — это состояние, в котором автомат находится в момент запуска. На вход автомата подаются символы некоторого алфавита. В зависимости от поданного на вход символа и текущего состояния автомат либо переходит в другое состояние, либо остаётся в текущем состоянии. Если пара (состояние, входной символ) однозначно определяет используемое правило перехода, то автомат называется **детерминированным**.

Дадим теперь формальное определение конечного автомата.

Определение 1.3.1. ***Конечным автоматом*** *(сокращённо КА) называется пятёрка $M(Q, \Sigma, \delta, q_0, F)$, в которой*

Q — конечное множество состояний автомата;

Σ — конечное множество допустимых входных символов (алфавит автомата);

δ — функция переходов, отображающая произведение $Q \times \Sigma$ во множество всех подмножеств множества Q, то есть $\delta : Q \times \Sigma \to \mathcal{P}(Q)$;

$q_0 \in Q$ — начальное состояние автомата;

F — непустое множество конечных состояний, $F \subseteq Q$, $F \neq \emptyset$.

Если функция δ определена на всём множестве $Q \times \Sigma$, то конечный автомат называется ***полностью определённым****.*

Если при всех $(q, a) \in Q \times \Sigma$ имеется не более одного состояния, в которое переходит КА, то автомат называется ***детерминированным*** *(сокращённо — ДКА). В противном случае автомат называется* ***недетерминированным*** *(сокращённо — НКА). Кроме того, у НКА могут быть переходы по ε.*

Функцию переходов δ можно задать таблицей. Такая таблица называется **таблицей переходов**. Ниже приведён пример таблицы переходов.

Таблица 1.3.1. Пример таблицы переходов для ДКА.

	a	b	c	Примечание
1	5		1	
2		4	1	конечное состояние
3			1	начальное состояние
4	3			
5	2			

В этой таблице в первом столбце указаны имена состояний автомата (в данном случае — просто числа), а a, b, c — символы, из которых состоит алфавит автомата. На пересечении строки с именем состояния и столбца, соответствующего символу алфавита, указано состояние, в которое переходит автомат. Например, на пересечении строки с именем 2 и столбца с именем b стоит 4. Следовательно, $\delta(2, b) = 4$. Если какая–либо ячейка пуста, то это означает, что соответствующего перехода нет.

Кроме того, переходы автомата можно задать с помощью ориентированного графа, вершинами которого являются состояния, а рёбрами — переходы. Такой граф называется **диаграммой переходов** конечного автомата. На следующем рисунке изображён пример диаграммы переходов.

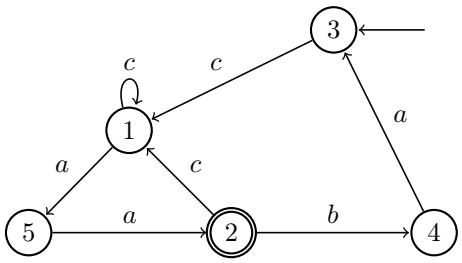


Рис. 1.3.1. Пример диаграммы переходов ДКА.

Здесь круги обозначают состояния, причём круги с двойной рамкой — конечные состояния; надпись над стрелкой обозначает символ, по которому совершается переход; переход выполняется из состояния в начале стрелки в состояние в конце стрелки. Стрелкой, не идущей ни из какого состояния, помечено начальное состояние автомата.

1.3.1. Преобразование недетерминированного автомата в детерминированный

Из определения конечного автомата следует, что любой ДКА является НКА. Обратное, вообще говоря, неверно. Однако доказано (см. [**?**, **?**]), что любой НКА можно преобразовать в распознающий тот же язык ДКА.

Идея преобразования состоит в том, что каждое состояние строящегося ДКА соответствует множеству состояний исходного, недетерминированного, автомата. После чтения входной строки $a_1 a_2 \dots a_n$ построенный ДКА находится в состоянии, соответствующем множеству состояний, которых может достичь исходный автомат по пути, помеченному строкой $a_1 a_2 \dots a_n$. Возможна ситуация, когда количество состояний построенного ДКА экспоненциально зависит от количества состояний исходного, недетерминированного, автомата. Однако при лексическом анализе реальных языков такого не бывает.

Прежде чем сформулировать алгоритм преобразования НКА в ДКА, с помощью приводимой ниже таблицы опишем необходимые для этого алгоритма операции.

Приведём теперь сам алгоритм построения ДКА по НКА.

Алгоритм 1.3.1. *Построение ДКА по НКА.*

Таблица 1.3.2. Операции для алгоритма преобразования НКА в ДКА.

Операция	Описание
ε–замыкание(s)	Множество состояний НКА, достижимых из состояния s по ε–переходам. При этом всегда $s \in \varepsilon$–замыкание(s).
ε–замыкание(T)	$\bigcup_{s \in T} \varepsilon$–замыкание($s$) ($T$ — множество состояний)
переход(T, a)	Множество состояний НКА, в которые имеется переход из некоторого состояния $s \in T$ по символу a.

Вход: *НКА* $M(Q, \Sigma, \delta, q_0, F)$.
Выход: *ДКА* $M'(Q', \Sigma, \delta', q_0', F')$.
Метод.

Изначально в Q' *имеется лишь одно состояние,* ε*–замыкание*(q_0)*, и оно не помечено. Далее делаем так:*

пока *в* Q *есть непомеченное состояние* T
  пометить T
  для всех $a \in \Sigma$
    $U \leftarrow \varepsilon$*–замыкание*(*переход*(T, a))
    если $U \notin Q'$ ***то***
      добавить U *в* Q' *как непомеченное состояние*
      положить $\delta'(T, a) = U$
    всё
  конец для
конец пока

Вычисление ε*–замыкания множества состояний* T *производится следующим образом:*

поместить все состояния множества T *в стек stack*
инициализировать ε*–замыкание*(T) *множеством* T
пока *stack не пуст*
  снять со стека верхний элемент, t
  для всех *состояний* u *с дугой от* t *к* u*, помеченной* ε
    если $u \notin \varepsilon$*–замыкание*(T) ***то***
      добавить u *во множество* ε*–замыкание*(T)
      поместить u *в stack*
    всё
  конец для
конец пока

Допускающими состояниями построенного автомата будут те состояния $T \in Q'$*, для которых* $T \cap F \neq \emptyset$.

Приведём пример применения этого алгоритма.

Пример 1.3.1. *Рассмотрим следующий недетерминированный КА, допускающий язык* $(a|b)^*abb$ *(*$\Sigma = \{a, b\}$*):*

Построим по этому НКА, пользуясь алгоритмом 1.3.1, соответствующий ДКА.

Прежде всего, начальным состоянием будет $A = \varepsilon$*–замыкание*$(0) = \{0, 1, 2, 4, 7\}$*. Вычислим* $\delta'(A, a) \equiv \varepsilon$*–замыкание*(*переход*$(A, a)$)*.*

Среди состояний $\{0, 1, 2, 4, 7\}$ *только у состояний 2 и 7 есть переход по символу* a *(в состояния 3 и 8 соответственно). Поэтому переход*$(A, a) = \{3, 8\}$*. Далее, из состояния 3 можно с помощью* ε*–переходов дойти в состояния 3, 6, 7, 1, 2, 4; а из состояния 8 — только в состояние 8. Поэтому* ε*–замыкание*(*переход*$(A, a)) = \{1, 2, 3, 4, 6, 7, 8\}$*. Обозначим это множество* B*. Поскольку конечное состояние НКА, 10, множеству* B *не принадлежит, то* B *конечным состоянием не является.*

Вычислим $\delta'(A, b) \equiv \varepsilon$*–замыкание*(*переход*$(A, b)$)*. Среди всех состояний множества* A *только у состояния 4 есть переход по символу* b*. Это переход в состояние 5. А из состояния 5 по* ε*–переходам можно дойти в состояния 5, 6, 7, 1, 2, 4. Значит* $\delta'(A, b) \equiv \varepsilon$*–замыкание*(*переход*$(A, b)) = \varepsilon$*–замыкание*$(\{5\}) = \{1, 2, 4, 5, 6, 7\}$*. Обозначим данное множество* C*.*

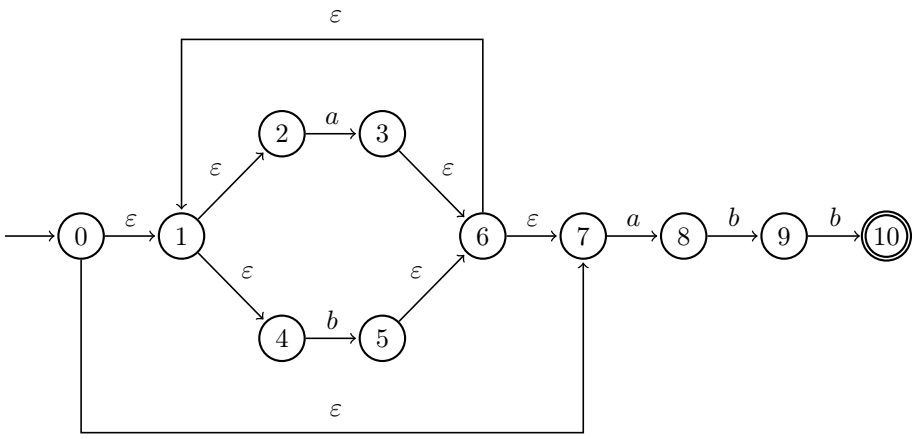


Рис. 1.3.2. Диаграмма переходов НКА, допускающего язык $(a|b)^*abb$.

Последующие выкладки выглядят так:

$$\begin{aligned}
\delta'(B,a) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(B,a)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,3,4,6,7,8\},a)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{2,7\},a)) = \varepsilon\text{–}\textit{замыкание}(\{3,8\}) = B;\\
\delta'(B,b) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(B,b)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,3,4,6,7,8\},b)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{4,8\},b)) = \varepsilon\text{–}\textit{замыкание}(\{5,9\}) = \{1,2,4,5,6,7,9\} \equiv D;\\
\delta'(C,a) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(C,a)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7\},a)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{2,7\},a)) = B;\\
\delta'(C,b) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(C,b)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7\},b)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{4\},b)) = \varepsilon\text{–}\textit{замыкание}(\{5\}) = C;\\
\delta'(D,a) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(D,a)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7,9\},a)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{2,7\},a)) = B;\\
\delta'(D,b) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(D,b)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7,9\},b)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{4,9\},a)) = \varepsilon\text{–}\textit{замыкание}(\{5,10\}) = \{1,2,4,5,6,7,10\} \equiv E;\\
\delta'(E,a) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(E,a)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7,10\},a)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{2,7\},a)) = B;\\
\delta'(E,b) &= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(E,b)) = \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{1,2,4,5,6,7,10\},b)) = \\
&= \varepsilon\text{–}\textit{замыкание}(\textit{переход}(\{4\},b)) = C
\end{aligned}$$

Соберём эти результаты в таблице переходов:

Таблица 1.3.3. Таблица переходов ДКА, распознающего язык $(a|b)^*abb$.

Множество состояний НКА	Состояние ДКА	a	b	Примечание
$\{0,1,2,4,7\}$	A	B	C	начальное состояние
$\{1,2,3,4,6,7,8\}$	B	B	D	
$\{1,2,4,5,6,7\}$	C	B	C	
$\{1,2,4,5,6,7,9\}$	D	B	E	
$\{1,2,4,5,6,7,10\}$	E	B	C	конечное состояние

Приведём диаграмму переходов, построенную по этой таблице:

1.3.2. Моделирование конечного автомата

Приведём теперь алгоритмы работы конечных автоматов. Начнём с детерминированных автоматов.

Алгоритм 1.3.2. *Моделирование ДКА.*

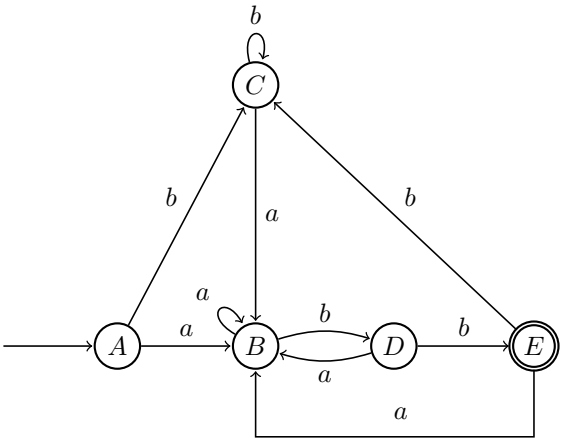


Рис. 1.3.3. Диаграмма переходов ДКА, допускающего язык $(a|b)^*abb$.

Вход: *входная строка x с завершающим символом eof и ДКА M с начальным состоянием q_0, набором принимающих состояний F и функцией переходов δ.*

Выход: *ответ „да“, если автомат M принимает строку x, и ответ „нет“ — в противном случае.*

Метод.

$s \leftarrow q_0$
$c \leftarrow$ *следующий_символ*()
пока $c \neq eof$
    $s \leftarrow \delta(s, c)$
    $c \leftarrow$ *следующий_символ*()
конец пока
если $s \in F$ ***то***
    выдать „да“
иначе
    выдать „нет“
всё

А теперь приведём алгоритм работы НКА.

Алгоритм 1.3.3. *Моделирование НКА.*

Вход: *входная строка x с завершающим символом eof и НКА M с начальным состоянием q_0, набором принимающих состояний F и функцией переходов δ.*

Выход: *ответ „да“, если автомат M принимает строку x, и ответ „нет“ — в противном случае.*

Метод.

$S \leftarrow \varepsilon$–*замыкание*(q_0)
$c \leftarrow$ *следующий_символ*()
пока $c \neq eof$
    $S \leftarrow \varepsilon$–*замыкание*(*переход*(S, c))
    $c \leftarrow$ *следующий_символ*()
конец пока
если $S \cap F \neq \emptyset$ ***то***
    выдать „да“
иначе
    выдать „нет“
всё

Приведённые алгоритмы моделирования работы конечных автоматов, по существу, являются алгоритмами, отвечающими на вопрос: принадлежит ли входная строка языку, задаваемому конечным автоматом, или нет. Однако для реализации лексического анализа удобнее одновременно с переходами по состояниям совершать действия по построению лексемы соответствующего типа.

1.3.3. Построение НКА по регулярному выражению

Приведём теперь алгоритм построения по произвольному регулярному выражению недетерминированного конечного автомата, распознающего язык, соответствующий регулярному выражению.

Алгоритм 1.3.4. *Алгоритм МакНотона–Ямады–Томпсона построения НКА по регулярному выражению.*

Вход: *регулярное выражение r над алфавитом Σ.*

Выход: *НКА N, принимающий язык $L(r)$.*

Метод.

Начнём с разбора выражения r на составляющие подвыражения. Правила построения НКА состоят из базисных правил для обработки подвыражений без операторов и индуктивных правил для построения бóльших конечных автоматов по автоматам для непосредственных подвыражений данного выражения. Далее действуем в соответствии со следующими правилами.

1) Для каждого подвыражения ε строим НКА

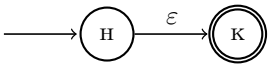


Рис. 1.3.4. НКА для подвыражения ε.

Здесь буквой „н“ обозначено новое состояние, представляющее собой начальное состояние НКА для подвыражения ε, а буквой „к“ — новое состояние, являющееся конечным состоянием НКА.

2) Для каждого подвыражения a, где a — символ алфавита Σ, строим НКА

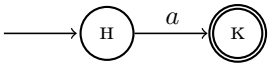


Рис. 1.3.5. НКА для подвыражения a, где a — символ алфавита Σ.

Здесь буквой „н“ обозначено новое состояние, представляющее собой начальное состояние НКА для подвыражения a, а буквой „к“ — новое состояние, являющееся конечным состоянием НКА.

При этом в обоих случаях для каждого подвыражения ε и каждого подвыражения a строится новый НКА, сколько бы в r ни было экземпляров ε и a.

3) Предположим теперь, что $N(s)$ и $N(t)$ — недетерминированные конечные автоматы, построенные по регулярным выражениям s и t соответственно.

а) Пусть $r = st$. В этом случае $N(r)$ строится так, как изображено на рис.1.3.6.

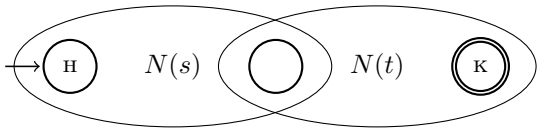


Рис. 1.3.6. НКА для выражения st.

Начальное состояние автомата $N(s)$ становится начальным состоянием автомата $N(r)$, а конечное состояние автомата $N(t)$ — единственным конечным состоянием автомата $N(r)$. Конечное состояние автомата $N(s)$ и начальное состояние автомата $N(t)$ склеиваются в одно состояние, со всеми входящими и исходящими переходами обоих состояний.

б) Пусть $r = s|t$. Тогда $N(r)$, НКА для выражения r, строится так, как показано на рис.1.3.7. Здесь буквой „н“ обозначено новое состояние, представляющее собой начальное состояние автомата $N(r)$, а буквой „к“ — новое состояние, являющееся конечным состоянием НКА. Обратите внимание, что принимающие состояния автоматов $N(s)$ и $N(t)$ не являются принимающими состояниями для $N(r)$.

в) Пусть $r = s^*$. Тогда для выражения r НКА $N(r)$ строится так, как изображено на рис.1.3.8.

г) Наконец, пусть $r = (s)$. Тогда $L(r) = L(s)$, так что в качестве $N(r)$ можно использовать $N(s)$.

Пример 1.3.2. *Построим с помощью сформулированного алгоритма НКА по выражению $r = (a|b)^*abb$. Как сказано в алгоритме 1.3.4, для построения НКА, соответствующего регулярному выражению, нужно последовательно строить автоматы для подвыражений, а затем данные автоматы склеивать.*

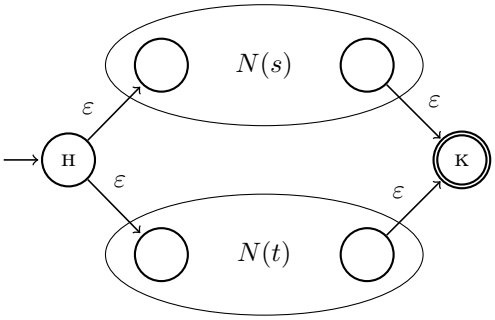


Рис. 1.3.7. НКА для выражения $s|t$.

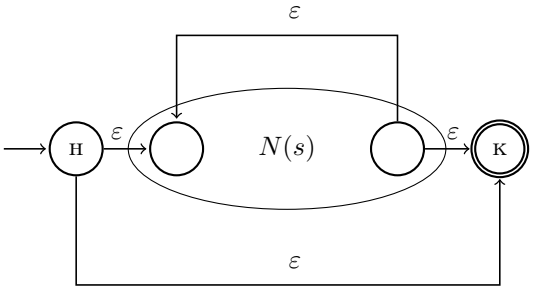


Рис. 1.3.8. НКА для выражения s^*.

Прежде всего рассмотрим подвыражение $r_1 = (a|b)$. *Согласно подпункту г) пункта 3 алгоритма 1.3.4, автомат* $N(r_1)$ *совпадает с автоматом для выражения* $r_2 = a|b$.

Построим автомат $N(r_2)$. *На основании подпункта б) пункта 3 алгоритма 1.3.4, автомат* $N(r_2)$ *конструируется из автоматов для подвыражений выражения* r_2, *то есть выражения* $r_3 = a$ *и выражения* $r_4 = b$. *В силу пункта 2) алгоритма 1.3.4, автоматы* $N(r_3)$ *и* $N(r_4)$ *имеют вид*

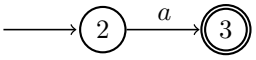


Рис. 1.3.9. НКА для выражения $r_3 = a$.

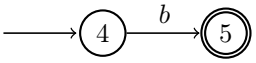
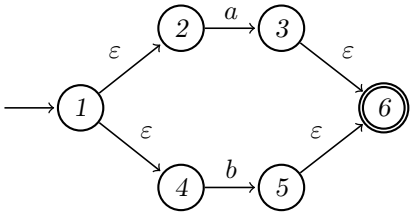


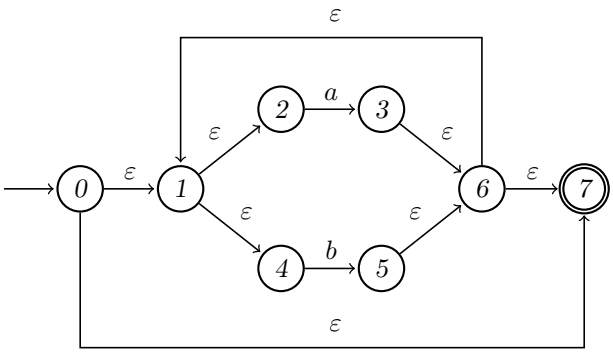
Рис. 1.3.10. НКА для выражения $r_4 = b$.

Построим теперь автомат $N(r_2)$, пользуясь подпунктом б) пункта 3 алгоритма 1.3.4:



Как уже сказано выше, автомат $N(r_1)$ совпадает с автоматом $N(r_2)$.

Рассмотрим подвыражение $r_5 = (a|b)^* = r_1^*$. Согласно подпункту в) пункта 3) алгоритма 1.3.4, автомат $N(r_5)$ выглядит так:



Построим теперь автомат для выражения $r_6 = (a|b)^*a$. Это выражение можно переписать в виде $r_6 = r_5 r_7$, где $r_7 = a$. Для построения $N(r_6)$ нужно построить автомат $N(r_7)$, а затем применить подпункт а) пункта 3) алгоритма 1.3.4:

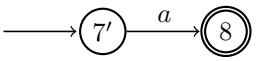


Рис. 1.3.11. НКА для выражения $r_7 = a$.

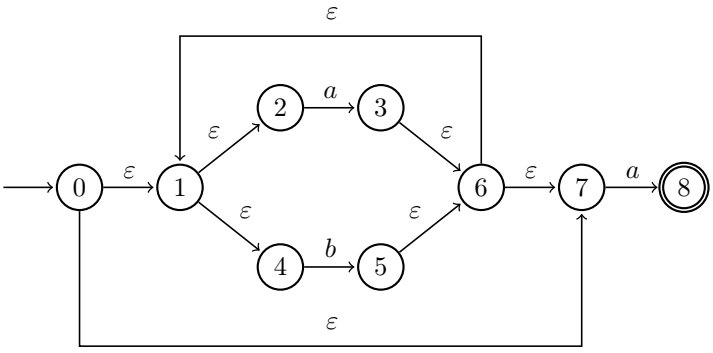


Рис. 1.3.12. НКА для выражения $r_6 = (a|b)^*a$.

Рассмотрим теперь подвыражение $r_8 = (a|b)^*ab$ выражения r. Перепишем выражение r_8 в виде $r_8 = r_6 r_9$, где $r_9 = b$. Чтобы построить $N(r_8)$, нужно построить $N(r_9)$, а затем отождествить конечное состояние автомата $N(r_6)$ с начальным состоянием автомата $N(r_9)$. В результате получим следующее:

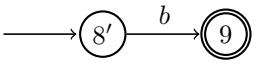


Рис. 1.3.13. НКА для выражения $r_9 = b$.

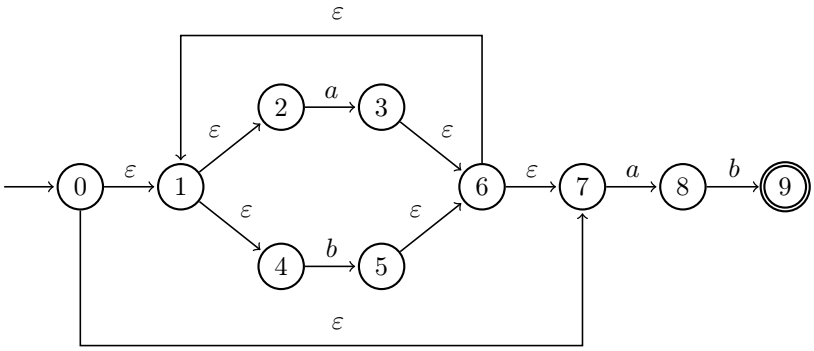


Рис. 1.3.14. НКА для выражения $r_8 = (a|b)^*ab$.

Наконец, рассмотрим само выражение $r = (a|b)^*abb$. Перепишем r в виде $r = r_8 r_{10}$, где $r_{10} = b$. Построим автомат для r_{10}: Автомат $N(r)$ получится из автоматов $N(r_8)$ и $N(r_{10})$, если отождествить начальное состояние автомата $N(r_{10})$ (состояние $9'$), с конечным состоянием автомата $N(r_8)$ (состояние 9). После такого отождествления получим следующую диаграмму переходов:

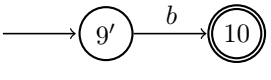


Рис. 1.3.15. НКА для выражения $r_{10} = b$.

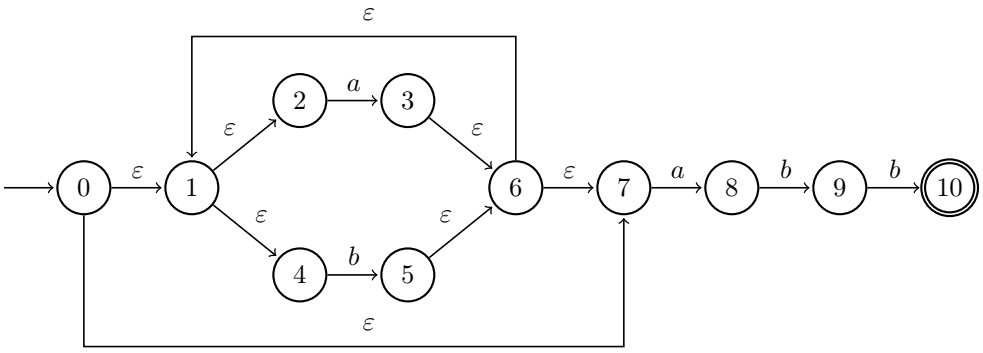


Рис. 1.3.16. НКА для выражения $r = (a|b)^*abb$.

1.3.4. Построение ДКА по регулярному выражению

В данном разделе будет сформулирован алгоритм построения детерминированного конечного автомата непосредственно по регулярному выражению, минуя стадию построения недетерминированного автомата.

Прежде чем сформулировать этот алгоритм, рассмотрим роли, которые играют разные состояния автоматов.

Назовём состояние недетерминированного конечного автомата **важным** если оно имеет исходящий переход не по ε.

Обратите внимание, что при вычислении множества состояний ε–замыкание(переход(T, a)), достижимых из T по входному символу a, используются только важные состояния. Таким образом,

множество состояний переход(s, a) — непусто, только если состояние s — важное. В процессе построения подмножеств два множества состояний недетерминированного автомата могут отождествляться, то есть рассматриваться как единое множество, если они

1) имеют одни и те же важные состояния;

2) либо оба содержат принимающие состояния, либо оба их не содержат.

При построении НКА по регулярному выражению важными состояниями являются только те, которые созданы как начальные для конкретных символов алфавита.

Построенный с помощью алгоритма 1.3.4 недетерминированный автомат имеет только одно принимающее состояние. Поскольку это состояние не имеет исходящих переходов, то оно важным не является. Приписав к регулярному выражению r справа уникальный ограничитель #[1], получим новое регулярное выражение, $(r)\#$, которое назовём **расширенным регулярным выражением**. Построив затем по расширенному выражению конечный автомат, получим, что любое состояние с переходом по символу # будет принимающим.

Важные состояния НКА соответствуют позициям в регулярном выражении, в которых находятся символы алфавита. Это удобно для представления регулярного выражения его **синтаксическим деревом**, в котором листья соответствуют операндам, а узлы — операторам. Внутренний узел называется **с–узлом**, **или–узлом**, или **звёздочка–узлом**, если он помечен соответственно оператором сцепления ($\circ$), объединения ($|$), или звёздочкой (*).

Пример 1.3.3. *На рис.1.3.17 изображено синтаксическое дерево для регулярного выражения $(a|b)^*abb\#$. С–узлы обозначены кружками.*

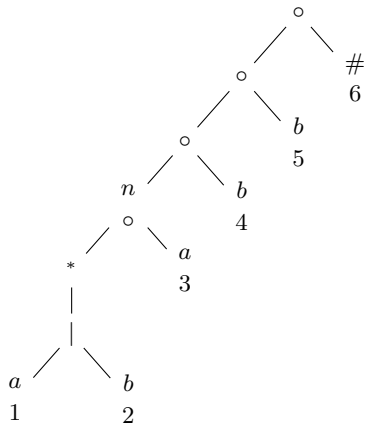


Рис. 1.3.17. Синтаксическое дерево для регулярного выражения $(a|b)^*abb\#$.

Листья синтаксического дерева помечаются символом ε или символами алфавита. Каждому листу, не помеченному ε, присваивается целочисленное значение, уникальное в пределах дерева. Это значение называется ***позицией*** *листа, а также позицией его символа. При этом каждый символ алфавита может иметь несколько позиций. Например, на рис.1.3.17 символ a имеет позиции 1 и 3. Иными словами, позиция символа алфавита, входящего в регулярное выражение, — это позиция символа в выражении, полученном из исходного выражения выбрасыванием скобок и операторов. Позиции в синтаксическом дереве соответствуют важным состояниям построенного НКА.*

Для построения ДКА непосредственно по регулярному выражению нужно определить функции, *зануляется*, *первые*, *последние*, *следующие*. Эти функции определяются так, как указано ниже, причём для их вычисления используется синтаксическое дерево для расширенного регулярного выражения $(r)\#$.

1) Значение *зануляется*(n) для узла n синтаксического дерева равно значению **истина** тогда и только тогда, когда подвыражение, представленное узлом n, содержит в своём языке ε. Иными словами, выражение может быть сделано пустой строкой, хотя может содержать в своём языке и непустые строки.

2) Значение *первые*(n) для узла n синтаксического дерева представляет собой множество позиций

[1]Уникальный в том смысле, что он не встречается во входном потоке.

в поддереве с корнем n, соответствующих первому символу как минимум одной строки в языке подвыражения с корнем n.

3) Значение *последние*(n) есть множество позиций в поддереве с корнем n, соответствующих последнему символу хотя бы одной строки в языке подвыражения с корнем n.

4) Значение *следующие*(p) для позиции p представляет собой множество позиций q в синтаксическом дереве в целом, для которых существует строка $x = a_1a_2\dots a_n$ языка $L((r)\#)$, такая, что для некоторого i символ a_i соответствует позиции p, а символ a_{i+1} — позиции q.

Пример 1.3.4. *Рассмотрим с–узел n на рис.1.3.17, соответствующий выражению $(a|b)^*a$. Тогда зануляется(n) =* **ложь**, *поскольку этот узел порождает все строки из a и b, оканчивающиеся на a, и не может порождать строки ε. С другой стороны, у звёздочка–узла ниже него значение функции зануляется равно значению* **истина**, *так как наряду со строками из a и b он может порождать и пустую строку, ε.*

Покажем теперь, что первые$(n) = \{1, 2, 3\}$. В са́мом деле, для любой строки вида aax первая позиция строки соответствует позиции 1 дерева, а первая позиция строки вида bax соответствует позиции 2. Однако если строка представляет собой a, то это a получается из позиции 3.

Далее, последние$(n) = \{3\}$, поскольку неважно, какая именно строка порождается по выражению для узла n — последняя позиция в строке представляет собой a, получающееся из позиции 3.

Наконец, вычислим следующие(1). Для этого рассмотрим строку вида $\dots ac\dots$, где c — символ алфавита, причём это a соответствует позиции 1, то есть является одним из символов, порождаемых a из подвыражения $(a|b)^*$. За этим a может следовать другое a или b из того же выражения, т.е. в этом случае c получается из позиций 1 и 2. Может также оказаться, что a — последнее в строке, порождённой выражением $(a|b)^*a$. Тогда символ c должен представлять собой a, получающееся из позиции 3. Таким образом, следующие$(1) = \{1, 2, 3\}$.

Опишем теперь, как вычислять четыре введённые функции. Что касается функций *зануляется*, *первые*, *последние*, то их можно вычислить рекурсией по высоте синтаксического дерева, применяя правила, указанные в табл.1.3.4, 1.3.5.

Таблица 1.3.4. Правила вычисления функций *зануляется* и *первые*.

Узел n	*зануляется*(n)	*первые*(n)
n — лист, помеченный ε	**истина**	$\emptyset$
n — лист с позицией i	**ложь**	$\{i\}$
или–узел $n = c_1\|c_2$	*зануляется*(c_1) **или** *зануляется*(c_2)	*первые*$(c_1) \cup$ *первые*(c_2)
с–узел $n = c_1c_2$	*зануляется*(c_1) **и** *зануляется*(c_2)	**если** *зануляется*(c_1) **то** *первые*$(c_1) \cup$ *первые*(c_2) **иначе** *первые*(c_1) **всё**
звёздочка–узел $n = c^*$	**истина**	*первые*(c)
узел $n = c?$	**истина**	*первые*(c)
узел $n = c^+$	*зануляется*(c)	*первые*(c)

Пример 1.3.5. *Из всех узлов на рис.1.3.17 функция зануляется равна* **истина** *только для звёздочка–узла. Поясним, почему. Из табл.1.3.4 видно, что ни для какого листа значение функции зануляется не равно* **истина**, *поскольку ни один лист не соответствует операнду, равному ε. Или–узел также не может дать значения* **истина**, *ибо ни один из его дочерних узлов не даёт этого значения. Звёздочка–узел имеет значение* **истина**, *так как это свойство любого звёздочка–узла. Наконец, в с–узле функция зануляется принимает значение* **ложь**, *если таково её значение хотя бы в одном из дочерних узлов. На рис.1.3.18 показано вычисление функций первые и последние для каждого из узлов (значение первые(n) показано слева от узла n, а значение последние(n) — справа).*

Каждый лист в качестве значений первые и последние, в соответствии с правилом для не–ε–узлов, имеет множество, состоящее только из него самого. Значения функций для или–узлов представляют собой объединение значений в дочерних узлах. Что же касается звёздочка–узла, то значения в нём функций первые и последние совпадают со значениями в единственном дочернем узле.

Таблица 1.3.5. Правила вычисления функции *последние*.

Узел n	*последние*(n)
n — лист, помеченный ε	$\emptyset$
n — лист с позицией i	$\{i\}$
или–узел $n = c_1 \vert c_2$	*последние*$(c_1) \cup$ *последние*(c_2)
с–узел $n = c_1 c_2$	**если** *зануляется*(c_2) **то** *последние*$(c_1) \cup$ *последние*(c_2) **иначе** *последние*(c_2) **всё**
звёздочка–узел $n = c^*$	*последние*(c)
узел $n = c?$	*последние*(c)
узел $n = c^+$	*последние*(c)

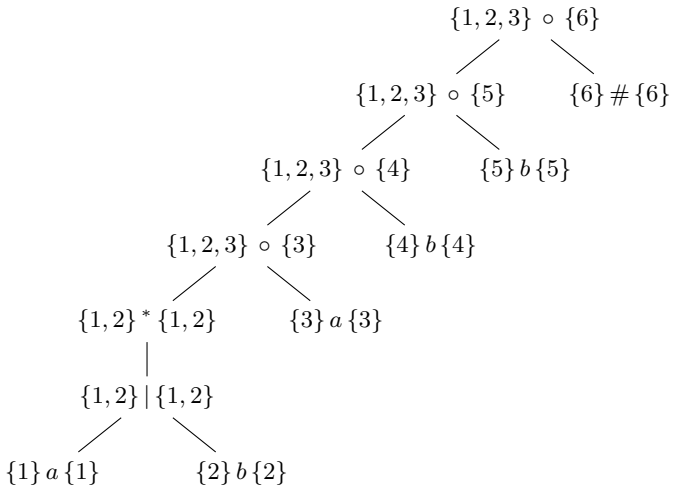


Рис. 1.3.18. Значения функций *первые* и *последние* в узлах синтаксического дерева для регулярного выражения $(a|b)^*abb\#$.

Перейдём теперь к са́мому нижнему с–узлу, обозначив его n. Вычисление первые(n) начнём с проверки значения зануляется в левом дочернем узле. В нашем случае оно равно **истина**. *Поэтому первые в узле n представляет собой объединение значений первые в дочерних узлах, т.е. равно* $\{1,2\} \cup \{3\} \equiv \{1,2,3\}$.

Перейдём теперь к вычислению функции *следующие*. Вычисляется эта функция по следующим двум правилам:

1) если n — с–узел с левым потомком c_1 и правым потомком c_2, то для каждой из позиций i из *последние*(c_1) все позиции из *первые*(c_2) содержатся в *следующие*(i);

2) если n — звёздочка–узел и i — позиция из *последние*(n), то все позиции из *первые*(n) содержатся в *следующие*(i).

Пример 1.3.6. *Вычислим теперь для узлов дерева, изображённого на рис.1.3.17, значение функции следующие. При этом, поскольку значения функций первые и последние для каждого узла изображены на рис.1.3.18, то для вычислений будем использовать рис.1.3.18. Согласно первому правилу, нужно просмотреть каждый с–узел и поместить каждую позицию из первые его правого дочернего узла в следующие для каждой позиции из последние его левого дочернего узла. Для са́мого нижнего с–узла на рис.1.3.18 это означает, что позиция 3 находится как следующие*(1), *так и в следующие*(2). *Рассмотрев следующий, находящийся выше, с–узел, получим, что позиция 4 содержится в следующие*(3). *Что же касается оставшихся двух с–узлов, то 5 входит в следующие*(4), *а 6 — в следующие*(5).

Применим теперь к звёздочка–узлу правило 2. Это правило гласит, что позиции 1 и 2 находятся как в следующие(1), *так и в следующие*(2), *ибо для этого узла и первые, и последние равны* $\{1,2\}$.

Результаты всех этих вычислений собраны в табл.1.3.6

Таблица 1.3.6. Значения функции *следующие*.

Узел n	*следующие*(n)
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Сформулируем, наконец, алгоритм построения ДКА непосредственно по регулярному выражению.

Алгоритм 1.3.5. *Построение ДКА по регулярному выражению.*
Вход: *регулярное выражение r над алфавитом Σ.*
Выход: *ДКА D, принимающий язык $L(r)$.*
Метод.

1) *Построить синтаксическое дерево T по расширенному регулярному выражению $(r)\#$.*

2) *Вычислить для дерева T функции зануляется, первые, последние, и следующие.*

3) *Построить $D_{сост}$ — множество состояний детерминированного конечного автомата D, и функцию $D_{п}$ — функцию переходов этого автомата, выполнив приводимую ниже процедуру. Состояния автомата D представляют собой множества позиций узлов дерева T. Изначально ни одно состояние „непомечено“; состояние становится „помеченным“ непосредственно перед тем, как рассматриваются его переходы. Начальным состоянием автомата D является первые(n_0), где n_0 — корень дерева T. Принимающими состояниями являются состояния, содержащие позицию для символа–ограничителя $\#$.*

```
алг состояния_и_переходы
    D_сост ← {первые(n_0)}        ▷ n_0 — корень дерева разбора для (r)#; единственное имеющееся
                                  ▷ состояние является начальным и непомечено
    пока в D_сост имеется непомеченное состояние S
        пометить S
        для всех входных символов a
            U ← ⋃_{p∈S, p отвечает a} следующие(p)
            если U ∉ D_сост то
                добавить U в D_сост как непомеченное
            всё
            положить D_п(S, a) = U
        конец для
    конец пока
кон
```

Пример 1.3.7. *Построим ДКА по регулярному выражению $r = (a|b)^*abb$. Синтаксическое дерево для $(r)\#$ показано на рис.1.3.17. Мы уже знаем, что функция зануляется имеет значение* **истина** *только в звёздочка–узле. Значения функций первые и последние показаны на рис.1.3.18, а значения функции следующие — в табл.1.3.6.*

Значение функции первые в корне равно $\{1,2,3\}$, так что это множество является начальным состоянием автомата $\mathcal{D}$. Обозначим это состояние A. Мы должны вычислить $\mathcal{D}_{п}(A,a)$ и $\mathcal{D}_{п}(A,b)$. Среди позиций из множества A символу a соответствуют позиции 1 и 3, а символу b — позиция 2. Таким образом, $\mathcal{D}_{п}(A,a) =$ следующие$(1) \cup$ следующие$(3) = \{1,2,3,4\}$, а $\mathcal{D}_{п}(A,b) =$ следующие$(2) = \{1,2,3\}$. Последнее состояние представляет собой состояние A, так что в $\mathcal{D}_{сост}$ его добавлять не нужно. Состояние же $B \equiv \{1,2,3,4\}$ является новым, так что добавляем его в $\mathcal{D}_{сост}$ и вычисляем

его переходы, $\mathcal{D}_п(B,a)$ и $\mathcal{D}_п(B,b)$:

$$\mathcal{D}_п(B,a) = \bigcup_{\substack{p\in B,\\ p\ \text{отвечает}\ a}} \textit{следующие}(p) = \textit{следующие}(1) \cup \textit{следующие}(3) = B;$$

$$\mathcal{D}_п(B,b) = \bigcup_{\substack{p\in B,\\ p\ \text{отвечает}\ b}} \textit{следующие}(p) = \textit{следующие}(2) \cup \textit{следующие}(4) = \{1,2,3\} \cup \{5\} = \{1,2,3,5\} \equiv C.$$

Вычислим переходы для состояния C:

$$\mathcal{D}_п(C,a) = \bigcup_{\substack{p\in C,\\ p\ \text{отвечает}\ a}} \textit{следующие}(p) = \textit{следующие}(1) \cup \textit{следующие}(3) = B;$$

$$\mathcal{D}_п(C,b) = \bigcup_{\substack{p\in C,\\ p\ \text{отвечает}\ b}} \textit{следующие}(p) = \textit{следующие}(2) \cup \textit{следующие}(5) = \{1,2,3\} \cup \{6\} = \{1,2,3,6\} \equiv D.$$

Состоянием, содержащим позицию, соответствующую символу #, является лишь состояние D. Следовательно, это состояние будет конечным.

Наконец, вычислим переходы для состояния D:

$$\mathcal{D}_п(D,a) = \bigcup_{\substack{p\in D,\\ p\ \text{отвечает}\ a}} \textit{следующие}(p) = \textit{следующие}(1) \cup \textit{следующие}(3) = B;$$

$$\mathcal{D}_п(D,b) = \bigcup_{\substack{p\in D,\\ p\ \text{отвечает}\ b}} \textit{следующие}(p) = \textit{следующие}(2) \cup \textit{следующие}(6) = \{1,2,3\} = A.$$

Соответствующая диаграмма переходов изображена на приводимом ниже рисунке.

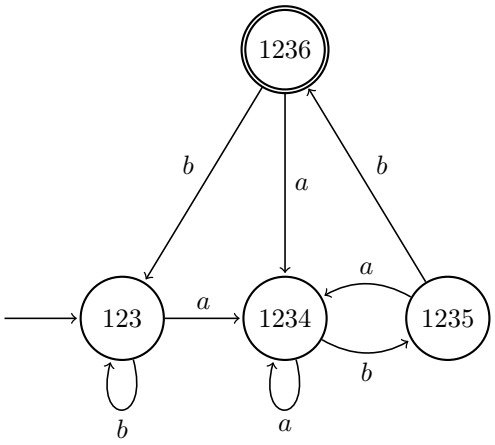


Рис. 1.3.19. Ещё один ДКА для регулярного выражения $r = (a|b)^*abb$.

1.3.5. Минимизация детерминированных конечных автоматов

Один и тот же язык могут распознавать разные ДКА. В качестве примера можно привести автоматы, диаграммы переходов которых изображены на рисунках 1.3.3 и 1.3.19. Такие автоматы могут иметь не только разные имена состояний, но и разное количество состояний. При реализации лексического анализатора с помощью ДКА, вообще говоря, лучше иметь конечный автомат с минимально возможным количеством состояний. Связано это с тем, что для каждого состояния нужны записи в таблице, описывающей лексический анализатор. При этом имена состояний значения не имеют.

Будем говорить, что два автомата **одинаковы с точностью до имён состояний**, если один из них можно получить из другого простым переименованием состояний. Однако между состояниями

автоматов, изображённых на рис.1.3.3 и 1.3.19 есть более тесная связь. А именно, состояния A и C автомата на рис.1.3.3 на самом деле эквивалентны, в том смысле, что ни одно из них не является принимающим, и любой входной символ приводит к одинаковым переходам — в состояние B для входного символа a и в состояние C для входного символа b. Если мы теперь склеим состояния A и C в одно состояние и обозначим получившееся состояние AC, то получим следующую диаграмму переходов:

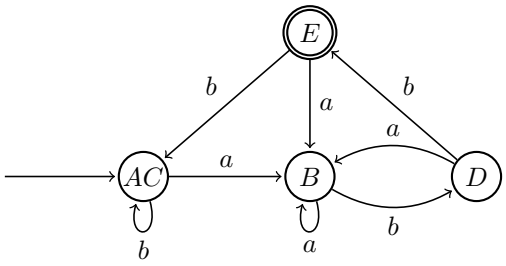


Рис. 1.3.20. Минимизированный ДКА для регулярного выражения $(a|b)^*abb$

Иными словами, состояния A и C ведут себя так же, как и состояние 123 на рис.1.3.19; состояние B — состояние 1234; состояние D — как состояние 1235; а состояние E — как состояние 1236.

Оказывается, что для любого регулярного языка всегда существует единственный (с точностью до имён состояний) ДКА, распознающий этот язык и имеющий минимальное количество состояний. Более того, этот ДКА с минимальным количеством состояний можно построить по любому ДКА для того же самого языка, склеив эквивалентные состояния. Для языка $L((a|b)^*abb)$ ДКА с минимальным количеством состояний показан на рис.1.3.19, и этот автомат можно получить из ДКА, изображённого на рис.1.3.3, сгруппировав состояниям следующим образом: $\{A, C\}\{B\}\{D\}\{E\}$.

Чтобы понять, как работает алгоритм преобразования ДКА в эквивалентный ДКА с минимальным количеством состояний, рассмотрим, как входные строки отличают одно состояние от другого. Будем говорить, что строка x **отличает** состояние s от состояния t, если ровно одно из состояний, достижимых из s и t, по пути с меткой x является принимающим, а другое — нет. Будем говорить, что состояние s **отличимо** от состояния t, если найдётся строка, которая их отличает.

Пример 1.3.8. *Пустая строка отличает любое принимающее состояние от непринимающего. На рис.1.3.3 строка bb отличает состояние A от состояния B, так как эта строка приводит из A в непринимающее состояние C, а из B — в принимающее состояние E.*

Алгоритм, минимизирующий количество состояний, работает путём разбиения множества состояний ДКА на группы неотличимых состояний. Каждая такая группа представляет собой одно состояние ДКА с минимальным количеством состояний. Алгоритм работает с разбиением, группы которого представляют собой множества состояний, отличие которых друг от друга пока не выявлено. Если никакую группу разбиения нельзя разделить на меньшие группы, то получен ДКА с минимальным количеством состояний.

Первоначально разбиение состоит из двух групп состояний: принимающие и непринимающие. Основной шаг состоит в том, чтобы взять некоторую группу состояний $A \equiv \{s_1, \ldots, s_k\}$ и некоторый входной символ a, и посмотреть, можно ли a использовать для отличения некоторых состояний в группе A. Делается это так: рассматриваем переходы из каждого из состояний $s_1, \ldots, s_k$ по символу a, и если состояния переходят в две или более группы текущего разбиения, то группу A разделяем так, чтобы состояния s_i и s_j оказывались в одной группе тогда и только тогда, когда они переходят в одну и ту же группу по данному входному символу a. Этот процесс продолжается до тех пор, пока не окажется ни одной группы, которую можно было бы разделить хотя бы одним входным символом. Данная идея реализована в следующем алгоритме.

Алгоритм 1.3.6. *Минимизация количества состояний ДКА.*

Вход: *ДКА D с множеством состояний Q, входным алфавитом Σ, начальным состоянием q_0 и множеством принимающих состояний F.*

Выход: *ДКА D', принимающий тот же язык, что и автомат D, и имеющий наименьшее возможное количество состояний.*

Метод.

1) *Начинаем с разбиения* Π *множества* Q *на две группы,* F *и* $Q \setminus F$*, состоящие, соответственно, из принимающих и непринимающих состояний автомата* D*.*

2) *Применяем приводимую ниже процедуру для построения нового разбиения,* $\Pi_{нов}$*.*

алг *новое_разбиение*(Π)
$\Pi_{нов} \leftarrow \Pi$
для всех $G \in \Pi_{нов}$
Разбиваем G *на подгруппы так, чтобы два состояния,* s *и* t*, находились в одной подгруппе тогда и только тогда, когда для каждого входного символа* a *состояния* s *и* t *имеют переходы по этому символу в состояния, принадлежащие одной и той же подгруппе группы* G*. При этом для разных символов подгруппы могут и отличаться. Заменяем в* $\Pi_{нов}$ *подгруппу* G *на набор построенных подгрупп.*
конец для
кон

3) *Если* $\Pi_{нов} = \Pi$*, то полагаем* $\Pi_{оконч} = \Pi$ *и переходим к шагу 4. В противном случае переходим к шагу 2, заменяя* Π *на* $\Pi_{нов}$*.*

4) *Выбираем из каждой группы разбиения* $\Pi_{оконч}$ *по одному состоянию в качестве* ***представителя*** *этой группы. Представители будут состояниями ДКА с минимальным количеством состояний, автомата* D'*. Остальные компоненты автомата* D' *строятся следующим образом.*

а) Начальное состояние автомата D' *— представитель группы, содержащей начальное состояние автомата* D*.*

б) Принимающими состояниями автомата D' *являются представители групп, содержащих принимающие состояния автомата* D*. Заметим, что каждая группа содержит либо только принимающие состояния, либо только непринимающие, поскольку мы начинали работу с отделения этих классов состояний друг от друга, а при вычислении нового разбиения строятся группы состояний, являющиеся подгруппами уже построенных групп.*

в) Пусть s *— представитель некоторой группы* G *в* $\Pi_{оконч}$*, и пусть переход по входному символу* a *в автомате* D *ведёт из состояния* s *в состояние* t*. Пусть* r *— представитель группы* H*, в которую входит состояние* t*. Тогда в* D' *имеется переход из* s *в* r *по входному символу* a*. Заметим, что в* D *каждое состояние группы* G *должно при входном символе* a *переходить в некоторое состояние группы* H*, ибо иначе группа* G *была бы разделена.*

1.4. Нерегулярные языки и лемма о разрастании

В данном разделе доказывается, что не все языки являются регулярными. А именно, мы докажем, что регулярным не является язык $\{a^n b^n : n > 0\}$. Для этого потребуется следующая лемма, говорящая о том, чем „внутренне устройство“ регулярных языков отличается от устройства языков нерегулярных.

Лемма 1.4.1. *(О разрастании (или о накачке) для регулярных языков.) Пусть* L *— некоторый регулярный язык с бесконечным количеством элементов. Тогда существует такое число* N*, что любая строка языка* L*, длина которой не менее* N*, может быть представлена в виде* xyz*, где*

1) подстрока y *— непуста (*$y \neq \varepsilon$*);*

2) $|xy| \leqslant N$*;*

3) строки $xz, xyz, xyyz, \ldots, xy^n z, \ldots$*, принадлежат языку* L*.*

Покажем теперь, что язык $L = \{a^n b^n : n > 0\}$ — нерегулярен. Предположим, что это не так. Тогда должна выполняться лемма о разрастании. Поскольку эта лемма должна выполняться для любой строки, длина которой не меньше N (которое мы не знаем), то лемма должна выполняться и для строки

$$\xi = \underbrace{a \ldots a}_{N} \underbrace{b \ldots b}_{N}.$$

Ясно, что $|\xi| = 2N$. По лемме строку ξ можно записать в виде $\xi = xyz$, где $|xy| \leqslant N$, $y \neq \varepsilon$. Так как первые N символов строки ξ — это символы a, то подстрока y может состоять лишь из символов a.

Но любая попытка „накачки“ приведёт к тому, что количество символов a увеличится, а количество символов b остаётся неизменным. Иными словами, „накачанная“ строка языка L не принадлежит. Полученное противоречие доказывает нерегулярность языка L.

Глава 2. Лексический анализатор

2.1. Конфигурационный фаил

Создан для учебного процессора МУР128(https://github.com/gavr-vlad-s/mur128)

```
%scaner_name LexerScaner
%codes_type lexem_code

%ident_name Id

%token_fields
"unsigned __int128 int_val;
__float128 float_val;"

%class_members
"__int128 integer_part;
__int128 fractional_part;
size_t exponent;
ssize_t exp_sign;
size_t frac_part_num_digits;
bool is_float;
char32_t precision;"

%codes
sp,     bp,
r0,     r1,     r2,       r3,       r4,     r5,
r6,     r7,     r8,       r9,       r10,    r11,
r12,    r13,    r14,      r15,      r16,    r17,
r18,    r19,    r20,      r21,      r22,    r23,
r24,    r25,    r26,      r27,      r28,    r29,
r30,    r31,
f0,     f1,     f2,       f3,       f4,     f5,
f6,     f7,     f8,       f9,       f10,    f11,
f12,    f13,    f14,      f15,      f16,    f17,
f18,    f19,    f20,      f21,      f22,    f23,
f24,    f25,    f26,      f27,      f28,    f29,
f30,    f31,
addi,   addf,   subi,     subf,
muliu,  mulis,  mulf,
diviu,  divis,  divf,
modiu,  modis,  divmodiu, divmodis,
add,    or,     xor,      not,
andn,   orn,    xorn,
lshift, rshift, cmp,      jmp,      jmpr,   jmpn,
jmpnr,  jmpz,   jmpzr,
jmpp,   jmppr,  jmpnz,    jmpnzr,   jmpge,  jmpger,
jmple,  jmpler, call,     callr,    trap,   reti,
mov,    movu,   movs,
mov8u,  mov8s,  mov16u,   mov16s,
mov32u, mov32s, mov64u,   mov64s,
round,
Equal, Semicolon,  Comma,
```

```
Plus, Minus, Mul, Div, Mod,
Open_func, Close_func,
Open_round, Close_round,
Open_square, Close_square,
Integer, Single, Double, Extended, Quatro

%idents {[:Letter:]|[:letter:]|_}{[:Letter:]|[:letter:]|_|[:digits:]}

%impl_additions "
#include <quadmath>

size_t digit2int(char32_t ch) {
size_t v = ch - U'0';
return (v<=9)? v : (v&0b1101'1111) - 7;
}
__int128 setexp(char32_t ch) {
return (ch == '-')? -1 : 1;
}

__float128 lexem_code: build_float(){
return integer_part + fractional_part*powq(10,-frac_part_num_digits)+exp_sign*exponent;
}

lexem_code precision2code(char32_t ch){
switch (ch) {
case: 'S':
return Single;
break;
case: 'D':
return Double;
break;
case: 'E':
return Extended;
break;
case: 'Q':
return Quatro;
break;
default:
return Single;
break;
}
}
"

%keywords
"addi" : addi,          "addf" : addf,      "subi" : subi,
"subf" : subf,
"muliu" : muliu,        "mulis" : mulis,    "mulf" : mulf,
"diviu" : diviu,        "divis" : divis,    "divf" : divf,
"modiu" : modiu,        "modis" : modis,    "divmodiu" : divmodiu,
"divmodis" : divmodis,
"add" : add,            "or" : or,          "xor" : xor,
"not" : not,
"andn" : andn,          "orn" : orn,        "xorn" : xorn,
"lshift" : lshift,      "rshift" : rshift,  "cmp" : cmp,
"jmp" : jmp,            "jmpr" : jmpr,      "jmpn" : jmpn,
"jmpnr" : jmpnr,        "jmpz" : jmpz,      "jmpzr" : jmpzr,
```

```
"jmpp" : jmpp,          "jmppr" : jmppr,    "jmpnz" : jmpnz,
"jmpnzr" : jmpnzr,      "jmpge" : jmpge,    "jmpger" : jmpger,
"jmple" : jmple,        "jmpler" : jmpler,
"call" : call,          "callr" : callr,    "trap" : trap,
"reti" : reti,
"mov" : mov,            "movu" : movu,      "movs" : movs,
"mov8u" : mov8u,        "mov8s" : mov8s,    "mov16u" : mov16u,
"mov16s" : mov16s,
"mov32u" : mov32u,      "mov32s" : mov32s,  "mov64u" : mov64u,
"mov64s" : mov64s,
"round" : round

%delimiters
"=" : Equal, ";" : Semicolon, "," : Comma,
"+" : Plus, "-" : Minus, "*" : Mul, "/" : Div, "%": Mod,
"^" : xor, "|" : or, "&" : add, "~" : not,
"<<" : lshift, ">>" : rshift,
"{":Open_func,    "}":Close_func,
"(":Open_round,   ")":Close_round,
"[":Open_square,  "]":Close_square

%numbers "int_val = 0;
float_val = 0;
is_float = false;
integer_part = 0;
fractional_part = 0;
exponent = 1;
exp_sign = 1;
frac_part_num_digits = 0;
token.code = Integer;"
:
"
if(is_float){
token.float_val=build_float();
token.code = precision2code(precision);
} else {
token.int_val=integer_part;
token.code = Integer;
}"
%action addHexDigit "integer_part = (integer_part << 4) + digit2int(ch);"
%action addDecDigit "integer_part = integer_part * 10 + digit2int(ch);"
%action addBinDigit "integer_part = (integer_part << 1) + digit2int(ch);"
%action addOctDigit "integer_part = (integer_part << 3) + digit2int(ch);"
%action setIsFloat "is_float = true;"
%action addDigitToDegree "exponent = exponent * 10 + digit2int(ch);"
%action addDecToFrac "fractional_part = fractional_part / 10 + digit2int(ch); frac_part_num_digits +
%action setExpSign "exp_sign = setexp(ch);"
%action setPrecision "precision = ch; is_float = true;"

{[:digits:]$addDecDigit('?[:digits:]$addDecDigit)*(.$setIsFloat[:digits:]$addDecToFrac('?[:digits:]$
?(((S|s)|(D|d)|(E|e)|(Q|q))$setPrecision+|-$setExpSign?[:digits:]$addDigitToDegree('?[:digits:]$addD
0o[:odigits:]$addOctDigit('?[:odigits:]$addOctDigit)*|
0(b|B)[:bdigits:]$addBinDigit('?[:bdigits:]$addBinDigit)*|
0(x|X)[:xdigits:]$addHexDigit('?[:xdigits:]$addHexDigit)*}
```

2.2. Созданные файлы

Листининг файла lexerscaner.h

```
#ifndef LEXERSCANER_H
#define LEXERSCANER_H

#include "../include/abstract_scaner.h"
#include "../include/error_count.h"
#include "../include/location.h"
#include <string>"

enum lexem_code : unsigned short {
None,         Unknown,    sp,
bp,           r0,         r1,
r2,           r3,         r4,
r5,           r6,         r7,
r8,           r9,         r10,
r11,          r12,        r13,
r14,          r15,        r16,
r17,          r18,        r19,
r20,          r21,        r22,
r23,          r24,        r25,
r26,          r27,        r28,
r29,          r30,        r31,
f0,           f1,         f2,
f3,           f4,         f5,
f6,           f7,         f8,
f9,           f10,        f11,
f12,          f13,        f14,
f15,          f16,        f17,
f18,          f19,        f20,
f21,          f22,        f23,
f24,          f25,        f26,
f27,          f28,        f29,
f30,          f31,        addi,
addf,         subi,       subf,
muliu,        mulis,      mulf,
diviu,        divis,      divf,
modiu,        modis,      divmodiu,
divmodis,     add,        orr,
xorr,          nott,        andn,
orn,          xorn,       lshift,
rshift,       cmp,        jmp,
jmpr,         jmpn,       jmpnr,
jmpz,         jmpzr,      jmpp,
jmppr,        jmpnz,      jmpnzr,
jmpge,        jmpger,     jmple,
jmpler,       call,       callr,
trap,         reti,       mov,
movu,         movs,       mov8u,
mov8s,        mov16u,     mov16s,
mov32u,       mov32s,     mov64u,
mov64s,       round,      Equal,
Semicolon,    Comma,      Plus,
Minus,        Mul,        Div,
```

```
Mod,          Open_func,  Close_func,
Open_round,   Close_round,Open_square,
Close_square,Integer,     Single,
Double,       Extended,   Quatro
};

struct Lexem_info{
lexem_code code;
union{
size_t    ident_index;
unsigned __int128 int_val;
__float128 float_val;
};
};

class LexerScaner : public Scaner<Lexem_info> {
public:
LexerScaner() = default;
LexerScaner(Location* location, const Errors_and_tries& et) :
Scaner<Lexem_info>(location, et) {};
LexerScaner(const LexerScaner& orig) = default;
virtual ~LexerScaner() = default;
virtual Lexem_info current_lexem();
private:
enum Automaton_name{
A_start,     A_unknown, A_idKeyword,
A_delimiter, A_number
};
Automaton_name automaton; /* current automaton */

typedef bool (LexerScaner::*Automaton_proc)();
/* This is the type of pointer to the member function that implements the
* automaton that processes the lexeme. This function must return true if
* the lexeme is not yet parsed, and false otherwise. */

typedef void (LexerScaner::*Final_proc)();
/* And this is the type of the pointer to the member function that performs
* the necessary actions in the event of an unexpected end of the lexeme. */

static Automaton_proc procs[];
static Final_proc     finals[];

/* Lexeme processing functions: */
bool start_proc();     bool unknown_proc();
bool idkeyword_proc(); bool delimiter_proc();
bool number_proc();
/* functions for performing actions in case of an
* unexpected end of the token */
void none_proc();            void unknown_final_proc();
void idkeyword_final_proc(); void delimiter_final_proc();
void number_final_proc();
};
#endif
```

Листининг файла lexerscaner.cpp

```
#include "../include/lexerscaner.h"
```

```
#include "../include/get_init_state.h"
#include "../include/search_char.h"
#include "../include/belongs.h"
#include <set>
#include <string.h>
#include <vector>
#include "../include/operation_with_sets.h"
#include <quadmath.h>

size_t digit2int(char32_t ch) {
size_t v = ch - U'0';
return (v<=9)? v : (v&0b1101'1111) - 7;
};
__int128 setexp(char32_t ch) {
return (ch == '-')? -1 : 1;
}

__float128 lexem_code:: build_float(){
return integer_part + fractional_part*powq(10,-frac_part_num_digits)+exp_sign*exponent;
}

lexem_code precision2code(char32_t ch){
switch (ch) {
case: 'S':
return Single;
break;
case: 'D':
return Double;
break;
case: 'E':
return Extended;
break;
case: 'Q':
return Quatro;
break;
default:
return Single;
break;
}
}

LexerScaner::Automaton_proc LexerScaner::procs[] = {
&LexerScaner::start_proc(),     &LexerScaner::unknown_proc(),
&LexerScaner::idkeyword_proc(), &LexerScaner::delimiter_proc(),
&LexerScaner::number_proc()
};

LexerScaner::Final_proc LexerScaner::finals[] = {
&LexerScaner::none_proc(),            &LexerScaner::unknown_final_proc(),
&LexerScaner::idkeyword_final_proc(), &LexerScaner::delimiter_final_proc(),
&LexerScaner::number_final_proc()
};

enum Category {
SPACES,     DELIMITER_BEGIN,
NUMBER0,    NUMBER5,
NUMBER6,    NUMBER_BEGIN,
```

```
NUMBER1,    NUMBER2,
NUMBER3,    NUMBER4,
NUMBER7,    NUMBER8,
NUMBER9,    NUMBER10,
NUMBER11,   IDKEYWORD_BEGIN,
IDKEYWORD0, IDKEYWORD1,
IDKEYWORD2, IDKEYWORD3,
Other
};

static const std::map<char32_t, uint32_t> categories_table = {
{'\0', 1},      {'\X01', 1},    {'\X02', 1},    {'\X03', 1},
{'\X04', 1},    {'\X05', 1},    {'\X06', 1},    {'\a', 1},
{'\b', 1},      {'\t', 1},      {'\n', 1},      {'\v', 1},
{'\f', 1},      {'\r', 1},      {'\X0e', 1},    {'\X0f', 1},
{'\X10', 1},    {'\X11', 1},    {'\X12', 1},    {'\X13', 1},
{'\X14', 1},    {'\X15', 1},    {'\X16', 1},    {'\X17', 1},
{'\X18', 1},    {'\X19', 1},    {'\X1a', 1},    {'\X1b', 1},
{'\X1c', 1},    {'\X1d', 1},    {'\X1e', 1},    {'\X1f', 1},
{' ', 1},       {'%', 2},       {'&', 2},       {', 512},
{'(', 2},       {')', 2},       {'*', 2},       {'+', 2},
{',', 2},       {'-', 258},     {'.', 2048},    {'/', 2},
{'0', 263228},  {'1', 525432},  {'2', 525424},  {'3', 525424},
{'4', 525424},  {'5', 263280},  {'6', 525424},  {'7', 263280},
{'8', 525408},  {'9', 263264},  {';', 2},       {'<', 2},
{'=', 2},       {'>', 2},       {'A', 361472},  {'B', 365568},
{'C', 361472},  {'D', 361600},  {'E', 361600},  {'F', 361472},
{'G', 360448},  {'H', 360448},  {'I', 360448},  {'J', 360448},
{'K', 360448},  {'L', 360448},  {'M', 360448},  {'N', 360448},
{'O', 360448},  {'P', 360448},  {'Q', 360576},  {'R', 360448},
{'S', 360576},  {'T', 360448},  {'U', 360448},  {'V', 360448},
{'W', 360448},  {'X', 368640},  {'Y', 360448},  {'Z', 360448},
{'[', 2},       {']', 2},       {'^', 2},       {'_', 360448},
{'a', 689152},  {'b', 627712},  {'c', 427008},  {'d', 689280},
{'e', 623744},  {'f', 623616},  {'g', 622592},  {'h', 622592},
{'i', 622592},  {'j', 425984},  {'k', 360448},  {'l', 688128},
{'m', 688128},  {'n', 688128},  {'o', 704512},  {'p', 622592},
{'q', 360576},  {'r', 688128},  {'s', 688256},  {'t', 688128},
{'u', 622592},  {'v', 622592},  {'w', 360448},  {'x', 434176},
{'y', 360448},  {'z', 622592},  {'{', 2},       {'|', 2},
{'}', 2},       {'~', 2},       {'Ё', 360448}, {'А', 360448},
{'Б', 360448}, {'В', 360448}, {'Г', 360448}, {'Д', 360448},
{'Е', 360448}, {'Ж', 360448}, {'З', 360448}, {'И', 360448},
{'Й', 360448}, {'К', 360448}, {'Л', 360448}, {'М', 360448},
{'Н', 360448}, {'О', 360448}, {'П', 360448}, {'Р', 360448},
{'С', 360448}, {'Т', 360448}, {'У', 360448}, {'Ф', 360448},
{'Х', 360448}, {'Ц', 360448}, {'Ч', 360448}, {'Ш', 360448},
{'Щ', 360448}, {'Ъ', 360448}, {'Ы', 360448}, {'Ь', 360448},
{'Э', 360448}, {'Ю', 360448}, {'Я', 360448}, {'а', 360448},
{'б', 360448}, {'в', 360448}, {'г', 360448}, {'д', 360448},
{'е', 360448}, {'ж', 360448}, {'з', 360448}, {'и', 360448},
{'й', 360448}, {'к', 360448}, {'л', 360448}, {'м', 360448},
{'н', 360448}, {'о', 360448}, {'п', 360448}, {'р', 360448},
{'с', 360448}, {'т', 360448}, {'у', 360448}, {'ф', 360448},
{'х', 360448}, {'ц', 360448}, {'ч', 360448}, {'ш', 360448},
{'щ', 360448}, {'ъ', 360448}, {'ы', 360448}, {'ь', 360448},
{'э', 360448}, {'ю', 360448}, {'я', 360448}, {'ё', 360448}
```

```
};


uint64_t get_categories_set(char32_t c){
auto it = categories_table.find(c);
if(it != categories_table.end()){
return it->second;
}else{
return 1ULL << Other;
}
}
bool LexerScaner::start_proc(){
bool t = true;
state = -1;
/* For an automaton that processes a token, the state with the number (-1) is
* the state in which this automaton is initialized. */
if(belongs(SPACES, char_categories)){
loc->current_line += U'\n' == ch;
return t;
}
lexem_begin_line = loc->current_line;
if(belongs(DELIMITER_BEGIN, char_categories)){
(loc->pcurrent_char)--; automaton = A_delimiter;
state = -1;
return t;
}

if(belongs(NUMBER_BEGIN, char_categories)){
(loc->pcurrent_char)--; automaton = A_number;
state = 0;
int_val = 0;
float_val = 0;
is_float = false;
integer_part = 0;
fractional_part = 0;
exponent = 1;
exp_sign = 1;
frac_part_num_digits = 0;
token.code = Integer;
return t;
}

if(belongs(IDKEYWORD_BEGIN, char_categories)){
(loc->pcurrent_char)--; automaton = A_idKeyword;
state = 0;
return t;
}

automaton = A_unknown;
return t;
}

bool LexerScaner::unknown_proc(){
return belongs(Other, char_categories);
}

struct Keyword_list_elem{
```

```
std::u32string keyword;
lexem_code kw_code;
};

static const Keyword_list_elem kwlist[] = {
{U"add", add},           {U"addf", addf},
{U"addi", addi},         {U"andn", andn},
{U"call", call},         {U"callr", callr},
{U"cmp", cmp},           {U"divf", divf},
{U"divis", divis},       {U"diviu", diviu},
{U"divmodis", divmodis}, {U"divmodiu", divmodiu},
{U"jmp", jmp},           {U"jmpge", jmpge},
{U"jmpger", jmpger},     {U"jmple", jmple},
{U"jmpler", jmpler},     {U"jmpn", jmpn},
{U"jmpnr", jmpnr},       {U"jmpnz", jmpnz},
{U"jmpnzr", jmpnzr},     {U"jmpp", jmpp},
{U"jmppr", jmppr},       {U"jmpr", jmpr},
{U"jmpz", jmpz},         {U"jmpzr", jmpzr},
{U"lshift", lshift},     {U"modis", modis},
{U"modiu", modiu},       {U"mov", mov},
{U"mov16s", mov16s},     {U"mov16u", mov16u},
{U"mov32s", mov32s},     {U"mov32u", mov32u},
{U"mov64s", mov64s},     {U"mov64u", mov64u},
{U"mov8s", mov8s},       {U"mov8u", mov8u},
{U"movs", movs},         {U"movu", movu},
{U"mulf", mulf},         {U"mulis", mulis},
{U"muliu", muliu},       {U"not", not},
{U"or", orr},             {U"orn", orn},
{U"reti", reti},         {U"round", round},
{U"rshift", rshift},     {U"subf", subf},
{U"subi", subi},         {U"trap", trap},
{U"xor", xor},           {U"xorn", xorn}
};

#define NUM_OF_KEYWORDS 54

#define THERE_IS_NO_KEYWORD (-1)

static int search_keyword(const std::u32string& finded_keyword){
int result      = THERE_IS_NO_KEYWORD;
int low_bound   = 0;
int upper_bound = NUM_OF_KEYWORDS - 1;
int middle;
while(low_bound <= upper_bound){
middle             = (low_bound + upper_bound) / 2;
auto& curr_kw      = kwlist[middle].keyword;
int compare_result = finded_keyword.compare(curr_kw);
if(0 == compare_result){
return middle;
}
if(compare_result < 0){
upper_bound = middle - 1;
}else{
low_bound   = middle + 1;
}
}
return result;
```

```
}

static const std::set<size_t> final_states_for_idkeywords = {
1
};

bool LexerScaner::idkeyword_proc(){
bool t             = true;
bool there_is_jump = false;
switch(state){
case 0:
if(belongs(IDKEYWORD0, char_categories)){
state = 1;
there_is_jump = true;
}
else if(belongs(IDKEYWORD1, char_categories)){
buffer += ch;
state = 1;
there_is_jump = true;
}

break;
case 1:
if(belongs(IDKEYWORD2, char_categories)){
state = 1;
there_is_jump = true;
}
else if(belongs(IDKEYWORD3, char_categories)){
buffer += ch;
state = 1;
there_is_jump = true;
}

break;
default:
;
}

if(!there_is_jump){
t = false;
if(!is_elem(state, final_states_for_idkeywords)){
printf("At line %zu unexpectedly ended identifier or keyword.", loc->current_line);
en->increment_number_of_errors();
}

int search_result = search_keyword(buffer);
if(search_result != THERE_IS_NO_KEYWORD) {
token.code = kwlist[search_result].kw_code;
}
}

return t;
}

static const State_for_char init_table_for_delimiters[] ={
{7, U'%'},  {10, U'&'}, {18, U'('}, {19, U')'}, {5, U'*'},
{3, U'+'},  {2, U','},  {4, U'-'},  {6, U'/'},  {1, U';'},
```

```
{12, U'<'}, {0, U'='},  {14, U'>'}, {20, U'['}, {21, U']'},
{8, U'^'},  {16, U'{'}, {9, U'|'},  {17, U'}'}, {11, U'~'}
};

struct Elem {
/** A pointer to a string of characters that can be crossed. */
char32_t*       symbols;
/** A lexeme code. */
lexem_code code;
/** If the current character matches symbols[0], then the transition to the state
*  first_state;
*  if the current character matches symbols[1], then the transition to the state
*  first_state + 1;
*  if the current character matches symbols[2], then the transition to the state
*  first_state + 2, and so on. */
uint16_t        first_state;
};

static const Elem delim_jump_table[] = {
{const_cast<char32_t*>(U""), Equal, 0},
{const_cast<char32_t*>(U""), Semicolon, 0},
{const_cast<char32_t*>(U""), Comma, 0},
{const_cast<char32_t*>(U""), Plus, 0},
{const_cast<char32_t*>(U""), Minus, 0},
{const_cast<char32_t*>(U""), Mul, 0},
{const_cast<char32_t*>(U""), Div, 0},
{const_cast<char32_t*>(U""), Mod, 0},
{const_cast<char32_t*>(U""), xor, 0},
{const_cast<char32_t*>(U""), orr, 0},
{const_cast<char32_t*>(U""), add, 0},
{const_cast<char32_t*>(U""), not, 0},
{const_cast<char32_t*>(U"<"), Unknown, 13},
{const_cast<char32_t*>(U""), lshift, 0},
{const_cast<char32_t*>(U">"), Unknown, 15},
{const_cast<char32_t*>(U""), rshift, 0},
{const_cast<char32_t*>(U""), Open_func, 0},
{const_cast<char32_t*>(U""), Close_func, 0},
{const_cast<char32_t*>(U""), Open_round, 0},
{const_cast<char32_t*>(U""), Close_round, 0},
{const_cast<char32_t*>(U""), Open_square, 0},
{const_cast<char32_t*>(U""), Close_square, 0}
};

bool LexerScaner::delimiter_proc(){
bool t = false;
if(-1 == state){
state = get_init_state(ch, init_table_for_delimiters,
sizeof(init_table_for_delimiters)/sizeof(State_for_char));
token.code = delim_jump_table[state].code;
t = true;
return t;
}
Elem elem = delim_jump_table[state];
token.code = delim_jump_table[state].code;
int y = search_char(ch, elem.symbols);
if(y != THERE_IS_NO_CHAR){
state = elem.first_state + y; t = true;
```

```
}
return t;
}

static const std::set<size_t> final_states_for_numbers = {
1, 2,  3, 4, 5, 6, 7, 8,
9, 10
};

bool LexerScaner::number_proc(){
bool t             = true;
bool there_is_jump = false;
switch(state){
case 0:
if(belongs(NUMBER0, char_categories)){
integer_part = integer_part * 10 + digit2int(ch);
state = 10;
there_is_jump = true;
}
else if(belongs(NUMBER1, char_categories)){
integer_part = integer_part * 10 + digit2int(ch);
state = 9;
there_is_jump = true;
}

break;
case 1:
if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}

break;
case 2:
if(belongs(NUMBER3, char_categories)){
exp_sign = setexp(ch);
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
fractional_part = fractional_part / 10 + digit2int(ch); frac_part_num_digits += 1;
state = 8;
there_is_jump = true;
}
else if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}

break;
case 3:
if(belongs(NUMBER4, char_categories)){
state = 2;
there_is_jump = true;
}
```

```
else if(belongs(NUMBER3, char_categories)){
exp_sign = setexp(ch);
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
fractional_part = fractional_part / 10 + digit2int(ch); frac_part_num_digits += 1;
state = 8;
there_is_jump = true;
}
else if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}

break;
case 4:
if(belongs(NUMBER4, char_categories)){
state = 11;
there_is_jump = true;
}
else if(belongs(NUMBER5, char_categories)){
integer_part = (integer_part << 1) + digit2int(ch);
state = 4;
there_is_jump = true;
}

break;
case 5:
if(belongs(NUMBER4, char_categories)){
state = 12;
there_is_jump = true;
}
else if(belongs(NUMBER6, char_categories)){
integer_part = (integer_part << 3) + digit2int(ch);
state = 5;
there_is_jump = true;
}

break;
case 6:
if(belongs(NUMBER4, char_categories)){
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
exponent = exponent * 10 + digit2int(ch);
state = 6;
there_is_jump = true;
}

break;
case 7:
if(belongs(NUMBER4, char_categories)){
state = 16;
there_is_jump = true;
```

```
}
else if(belongs(NUMBER7, char_categories)){
integer_part = (integer_part << 4) + digit2int(ch);
state = 7;
there_is_jump = true;
}

break;
case 8:
if(belongs(NUMBER4, char_categories)){
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER3, char_categories)){
exp_sign = setexp(ch);
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
fractional_part = fractional_part / 10 + digit2int(ch); frac_part_num_digits += 1;
state = 8;
there_is_jump = true;
}
else if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}

break;
case 9:
if(belongs(NUMBER4, char_categories)){
state = 15;
there_is_jump = true;
}
else if(belongs(NUMBER3, char_categories)){
exp_sign = setexp(ch);
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER8, char_categories)){
is_float = true;
state = 13;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
integer_part = integer_part * 10 + digit2int(ch);
state = 9;
there_is_jump = true;
}
else if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}

break;
```

```
case 10:
if(belongs(NUMBER4, char_categories)){
state = 15;
there_is_jump = true;
}
else if(belongs(NUMBER3, char_categories)){
exp_sign = setexp(ch);
state = 14;
there_is_jump = true;
}
else if(belongs(NUMBER8, char_categories)){
is_float = true;
state = 13;
there_is_jump = true;
}
else if(belongs(NUMBER_BEGIN, char_categories)){
integer_part = integer_part * 10 + digit2int(ch);
state = 9;
there_is_jump = true;
}
else if(belongs(NUMBER9, char_categories)){
state = 11;
there_is_jump = true;
}
else if(belongs(NUMBER2, char_categories)){
precision = ch; is_float = true;
state = 1;
there_is_jump = true;
}
else if(belongs(NUMBER10, char_categories)){
state = 16;
there_is_jump = true;
}
else if(belongs(NUMBER11, char_categories)){
state = 12;
there_is_jump = true;
}

break;
case 11:
if(belongs(NUMBER5, char_categories)){
integer_part = (integer_part << 1) + digit2int(ch);
state = 4;
there_is_jump = true;
}

break;
case 12:
if(belongs(NUMBER6, char_categories)){
integer_part = (integer_part << 3) + digit2int(ch);
state = 5;
there_is_jump = true;
}

break;
case 13:
if(belongs(NUMBER_BEGIN, char_categories)){
```

```
fractional_part = fractional_part / 10 + digit2int(ch); frac_part_num_digits += 1;
state = 3;
there_is_jump = true;
}

break;
case 14:
if(belongs(NUMBER_BEGIN, char_categories)){
exponent = exponent * 10 + digit2int(ch);
state = 6;
there_is_jump = true;
}

break;
case 15:
if(belongs(NUMBER_BEGIN, char_categories)){
integer_part = integer_part * 10 + digit2int(ch);
state = 9;
there_is_jump = true;
}

break;
case 16:
if(belongs(NUMBER7, char_categories)){
integer_part = (integer_part << 4) + digit2int(ch);
state = 7;
there_is_jump = true;
}

break;
default:
;
}

if(!there_is_jump){
t = false;
if(!is_elem(state, final_states_for_numbers)){
printf("At line %zu unexpectedly ended the number.", loc->current_line);
en->increment_number_of_errors();
}

if(is_float){
token.float_val=build_float();
token.code = precision2code(precision);
} else {
token.int_val=integer_part;
token.code = Integer;
}
}

return t;
}

void LexerScaner::none_proc(){
/* This subroutine will be called if, after reading the input text, it turned
* out to be in the A_start automaton. Then you do not need to do anything. */
}
```

```
void LexerScaner::unknown_final_proc(){
/* This subroutine will be called if, after reading the input text, it turned
* out to be in the A_unknown automaton. Then you do not need to do anything. */
}

void LexerScaner::idkeyword_final_proc(){
if(!is_elem(state, final_states_for_idkeywords)){
printf("At line %zu unexpectedly ended identifier or keyword.", loc->current_line);
en->increment_number_of_errors();
}

int search_result = search_keyword(buffer);
if(search_result != THERE_IS_NO_KEYWORD) {
token.code = kwlist[search_result].kw_code;
}

}

void LexerScaner::delimiter_final_proc(){

token.code = delim_jump_table[state].code;

}

void LexerScaner::number_final_proc(){
if(!is_elem(state, final_states_for_numbers)){
printf("At line %zu unexpectedly ended the number.", loc->current_line);
en->increment_number_of_errors();
}

if(is_float){
token.float_val=build_float();
token.code = precision2code(precision);
} else {
token.int_val=integer_part;
token.code = Integer;
}
}

Lexem_info LexerScaner::current_lexem(){
automaton = A_start; token.code = None;
lexem_begin = loc->pcurrent_char;
bool t = true;
while((ch = *(loc->pcurrent_char)++)){
char_categories = get_categories_set(ch); //categories_table[ch];
t = (this->*procs[automaton])();
if(!t){
/* We get here only if the lexeme has already been read. At the same time,
* the current automaton reads the character immediately after the end of
* the token read, based on this symbol, it is decided that the token has
* been read and the transition to the next character has been made.
* Therefore, in order to not miss the first character of the next lexeme,
* we need to decrease the pcurrent_char pointer by one. */
(loc->pcurrent_char)--;
return token;
}
```

```
}
/* Here we can be, only if we have already read all the processed text. In this
* case, the pointer to the current symbol indicates a byte, which is immediately
* after the zero character, which is a sign of the end of the text. To avoid
* entering subsequent calls outside the text, we need to go back to the null
* character. */
(loc->pcurrent_char)--;
/* Further, since we are here, the end of the current token (perhaps unexpected)
* has not yet been processed. It is necessary to perform this processing, and,
* probably, to display any diagnostics. */
(this->*finals[automaton])();
return token;
}
```

Более подробную информацию можно узнать по репозиторуию на сервисе GitHub(https://github.com/ChikFTW/lexer)

Список литературы

[1] Зуев Е. Редкая профессия. – М.: ДМК Пресс, 2014.

[2] Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2014.

[3] Гаврилов В.С. Генератор лексических анализаторов Мяука, 2017.