Генератор лексических анализаторов Мяука
Версия 1.0.0

Гаврилов Владимир Сергеевич

13 ноября 2016 г.

Оглавление

Введение

Проект Мяука представляет собой генератор лексических анализаторов, порождающий текст лексического анализатора на языке C++. К данному моменту имеется достаточно много таких генераторов, например Coco/R, flex, flex++, flexc++, и этот список далеко не полон. Однако у всех этих генераторов есть один общий недостаток. Состоит указанный недостаток в том, что этими генераторами, по существу, автоматизируются лишь задачи проверки корректности записи и обнаружения начала лексем, а порождение значения лексемы по её строковому представлению должно выполняться вызываемой после проверки корректности лексемы функцией, написанной пользователем генератора. При этом, во–первых, дважды выполняется проход по фрагменту входной строки, и, во–вторых, приходится вручную реализовывать часть конечного автомата, построенного генератором лексических анализаторов. Предлагаемый генератор нацелен на устранение данного недостатка.

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) $\{\text{А}, \text{Б}, \text{В}, \text{Г}, \text{Д}, \text{Е}, \text{Ё}, \ldots, \text{Я}\}$ — 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 [2].

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 [3]. 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:*

$$\begin{aligned}
&\textit{буква} \to [A - Za - z]\\
&\textit{буква_или_подчерк} \to \textit{буква}|_\\
&\textit{цифра} \to [0 - 9]\\
&\textit{идентификатор} \to \textit{буква_или_подчерк}(\textit{буква_или_подчерк}|\textit{цифра})^*
\end{aligned}$$

Определение же беззнаковых чисел можно переписать так:

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

Упражнение 1.2.1. *Опишите языки, соответствующие следующим регулярным выражениям:*

а) $a(a|b)^*a$;

б) $((\varepsilon|a)b^*)^*$;

в) $(a|b)^*a(a|b)(a|b)$;

г) $a^*ba^*ba^*ba^*$;

д) $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$.

Упражнение 1.2.2. *Запишите классы символов для следующих множеств:*

а) первые десять букв русского алфавита, как в верхнем, так и в нижнем регистре;

б) строчные согласные русского алфавита;

в) цифры шестнадцатиричного числа (для цифр, бóльших 9, могут использоваться как строчные, так и прописные буквы);

г) символы, которые могут находиться в конце корректного предложения на русском языке.

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.

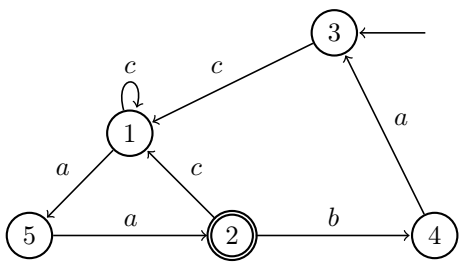


Рис. 1.3.1. Пример диаграммы переходов ДКА.

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. [1, 2]), 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 $a_1a_2\dots a_n$ построенный ДКА находится в состоянии, соответствующем множеству состояний, которых может достичь исходный автомат по пути, помеченному строкой $a_1a_2\dots a_n$. 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.

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)	$\bigcup_{s\in T}$ ε–замыкание(s) (T — множество состояний)
переход(T, a)	Множество состояний НКА, в которые имеется переход из некоторого состояния $s \in T$ по символу a.

Приведём теперь сам алгоритм построения ДКА по НКА.

Алгоритм 1.3.1. *Построение ДКА по НКА.*

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 $(a|b)^*abb$ ($\Sigma = \{a, b\}$):*

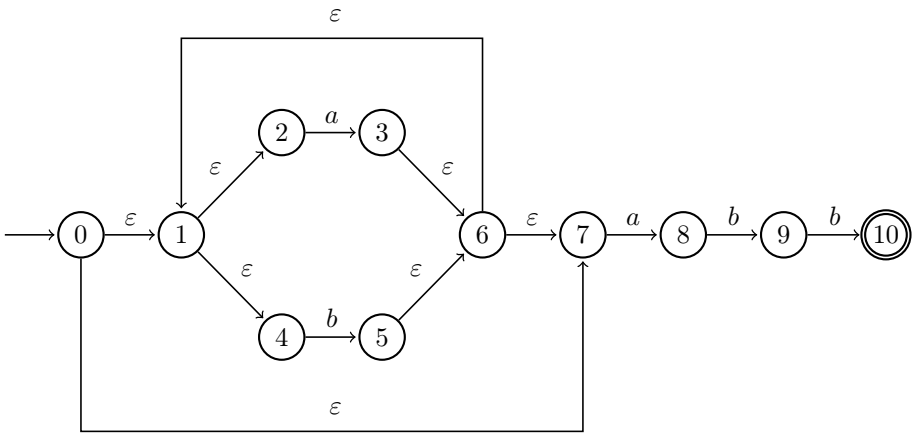


Рис. 1.3.2. Диаграмма переходов НКА, допускающего язык $(a|b)^*abb$.

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.

Последующие выкладки выглядят так:

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

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:

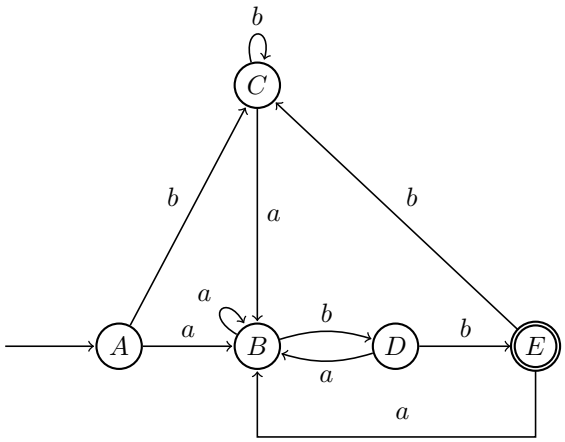


Рис. 1.3.3. Диаграмма переходов ДКА, допускающего язык $(a|b)^*abb$.

1.3.2. Моделирование конечного автомата

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

Алгоритм 1.3.2. *Моделирование ДКА.*

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

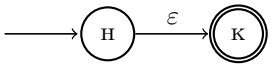


Рис. 1.3.4. НКА для подвыражения ε.

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

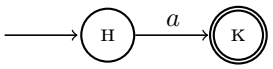


Рис. 1.3.5. НКА для подвыражения a, где a — символ алфавита Σ.

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.

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.

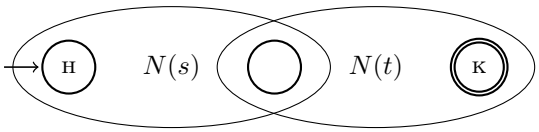


Рис. 1.3.6. НКА для выражения st.

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 $N(r)$.

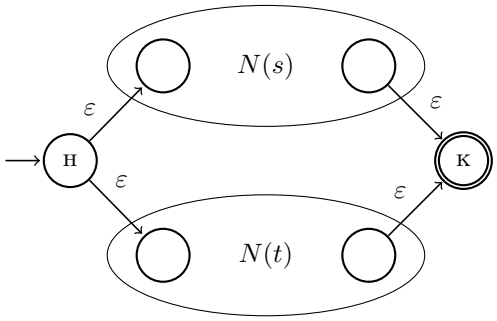


Рис. 1.3.7. НКА для выражения $s|t$.

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

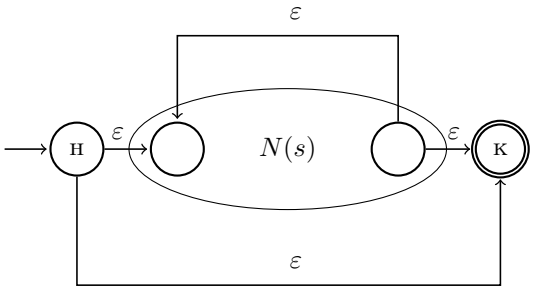


Рис. 1.3.8. НКА для выражения s^*.

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.*

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

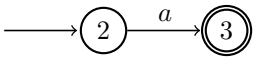


Рис. 1.3.9. НКА для выражения $r_3 = a$.

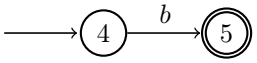
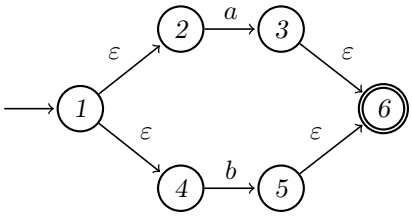


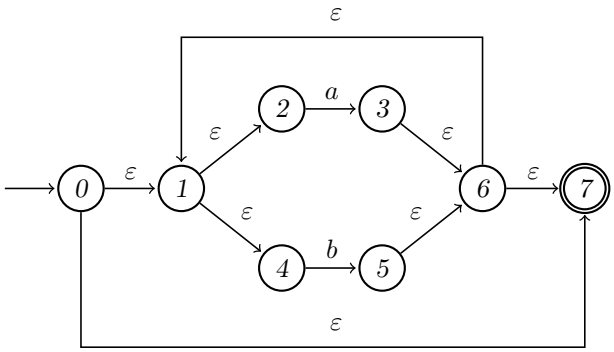
Рис. 1.3.10. НКА для выражения $r_4 = b$.

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:

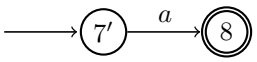


Рис. 1.3.11. НКА для выражения $r_7 = a$.

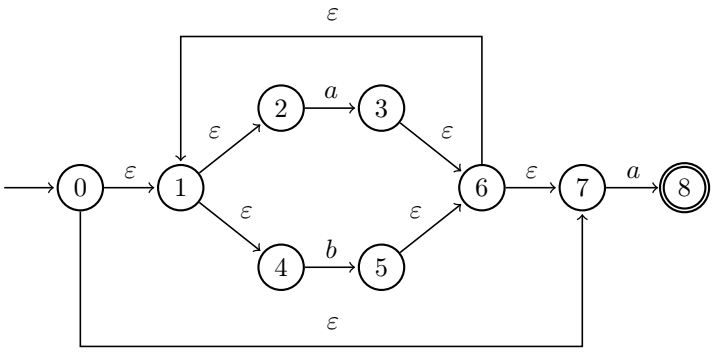


Рис. 1.3.12. НКА для выражения $r_6 = (a|b)^*a$.

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:

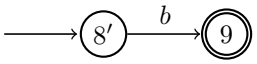


Рис. 1.3.13. НКА для выражения $r_9 = b$.

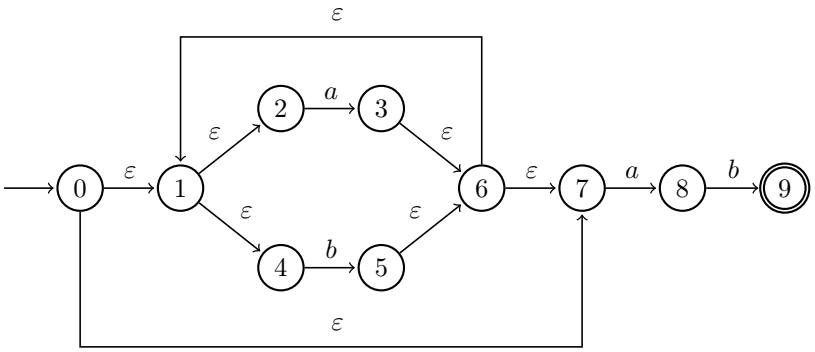


Рис. 1.3.14. НКА для выражения $r_8 = (a|b)^*ab$.

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 отожде-

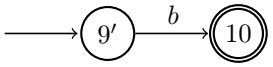


Рис. 1.3.15. НКА для выражения $r_{10} = b$.

ствить начальное состояние автомата $N(r_{10})$ (состояние $9'$), с конечным состоянием автомата $N(r_8)$ (состояние 9). После такого отождествления получим следующую диаграмму переходов:

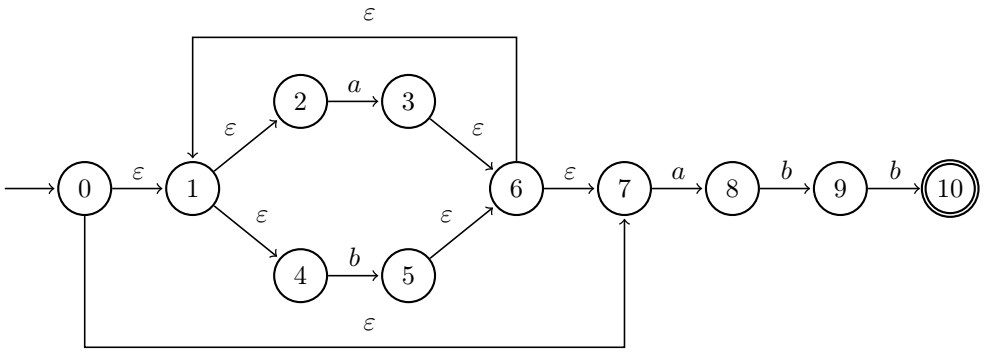
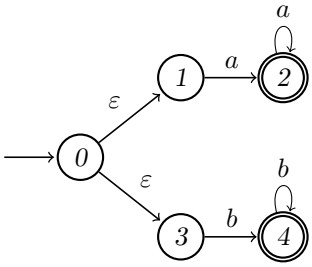


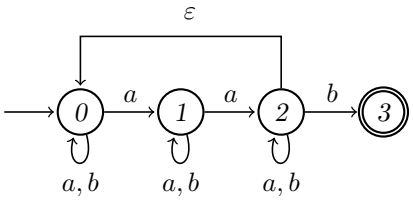
Рис. 1.3.16. НКА для выражения $r = (a|b)^*abb$.

Упражнение 1.3.1. *Преобразуйте к детерминированному виду следующие недетерминированные конечные автоматы:*

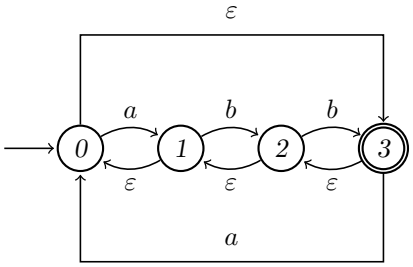
а) автомат



б) автомат



в) автомат



Упражнение 1.3.2. *Пользуясь алгоритмами 1.3.1 и 1.3.4, постройте детерминированные конечные автоматы по следующим регулярным выражениям:*

а) $(a|b)^*$;

б) $(a^*|b^*)$;

в) $((\varepsilon|a)|b^*)^*$;

г) $(a|b)^*abb(a|b)^*$.

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

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

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.*

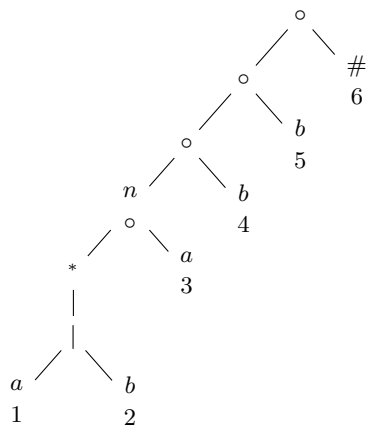


Рис. 1.3.17. Синтаксическое дерево для регулярного выражения $(a|b)^*abb\#$.

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 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 $x = a_1a_2\ldots a_n$ 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* $\ldots ac \ldots$, *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)

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 $n = c_1\|c_2$	*последние*$(c_1) \cup$ *последние*(c_2)
с–узел $n = c_1c_2$	**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)

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.*

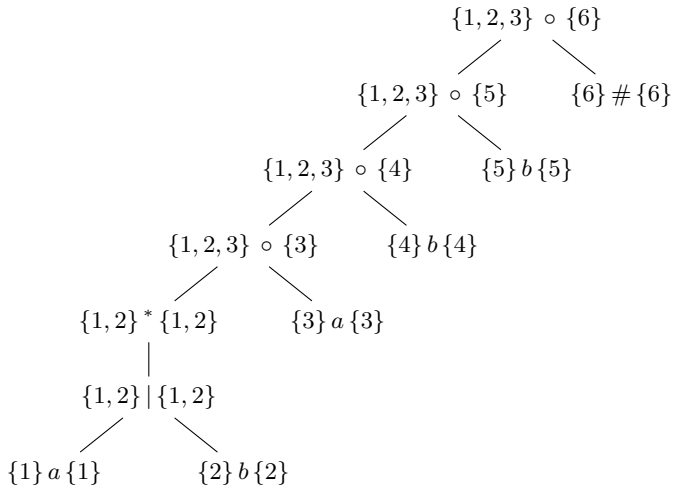


Рис. 1.3.18. Значения функций *первые* и *последние* в узлах синтаксического дерева для регулярного выражения $(a|b)^*abb\#$.

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*
  $D_{сост} \leftarrow \{первые(n_0)\}$ ▷ 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*
      $U \leftarrow \bigcup\limits_{\substack{p \in S,\\ p \text{ отвечает } a}} следующие(p)$
      если $U \notin D_{сост}$ ***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 $\mathcal{D}_{п}(B, b)$:

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

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

Вычислим переходы для состояния C:

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

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

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

Наконец, вычислим переходы для состояния D:

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

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

Соответствующая диаграмма переходов изображена на приводимом ниже рисунке.

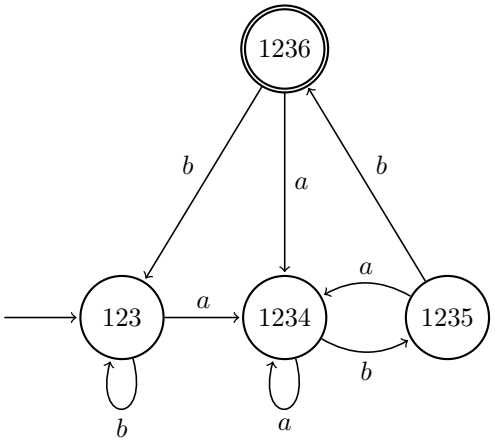


Рис. 1.3.19. Ещё один ДКА для регулярного выражения $r = (a|b)^*abb$.

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:

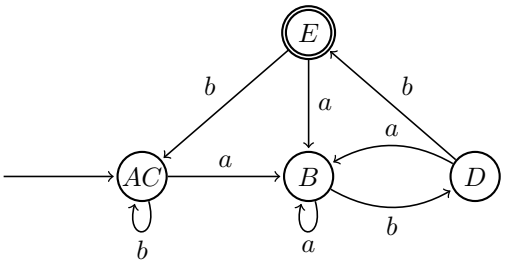


Рис. 1.3.20. Минимизированный ДКА для регулярного выражения $(a|b)^*abb$

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(П)
    П_нов ← П
    для всех G ∈ П_нов
        Разбиваем G на подгруппы так, чтобы два состояния, s и t, находились в
        одной подгруппе тогда и только тогда, когда для каждого входного сим-
        вола a состояния s и t имеют переходы по этому символу в состояния,
        принадлежащие одной и той же подгруппе группы G. При этом для раз-
        ных символов подгруппы могут и отличаться. Заменяем в П_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.*

Пример 1.3.9. *Снова вернёмся к ДКА на рис.1.3.3. Начальное разбиение состоит из двух групп,* $\{A, B, C, D\}\{E\}$*, состоящих, соответственно, из непринимающих и принимающего состояния. Для построения* $\Pi_{нов}$ *рассмотрим обе группы и входные символы* a *и* b*. Группу* $\{E\}$ *разделить нельзя, так как она состоит из одного состояния. Поэтому в* $\Pi_{нов}$ *эта группа остаётся неизменной.*

Группу $\{A, B, C, D\}$ *разделить можно, так что нужно рассмотреть воздействие на неё каждого входного символа. При входном символе* a *все состояния группы переходят в состояние* B*, так что отличить эти состояния при помощи строки, начинающейся с* a*, невозможно. При входном символе* b *состояния* A*,* B*, и* C *переходят в состояния, являющиеся членами группы* $\{A, B, C, D\}$*, тогда как состояние* D *переходит в* E*, которое является членом другой группы. Таким образом, в* $\Pi_{нов}$ *группа* $\{A, B, C, D\}$ *разбивается на* $\{A, B, C\}\{D\}$*, и* $\Pi_{нов}$ *на данной итерации представляет собой* $\{A, B, C\}\{D\}\{E\}$*.*

На следующей итерации группу $\{A, B, C\}$ *можно разбить на группы* $\{A, C\}\{B\}$*, поскольку и* A*, и* C *при входном символе* b *переходят в состояния группы* $\{A, B, C\}$*, а состояние* B *по этому входному символу переходит в состояние из другой группы,* $\{D\}$*. Таким образом, после второй итерации* $\Pi_{нов} = \{A, C\}\{B\}\{D\}\{E\}$*. На следующей итерации единственную оставшуюся группу с более чем одним состоянием разделить не получится, ибо и* A*, и* C *переходят в одно и то же состояние (а, значит, и в одну и ту же группу) по любому входному символу. Как следствие,* $\Pi_{оконч} = \{A, C\}\{B\}\{D\}\{E\}$*.*

Построим теперь по получившемуся разбиению компоненты ДКА с минимальным количеством состояний. Он имеет четыре состояния, соответствующие четырём группам разбиения $\Pi_{оконч}$*. В качестве представителей групп выбираем состояния* A*,* B*,* D*, и* E*. Начальным состоянием является состояние* A*, а единственным принимающим состоянием — состояние* E*. В табл.1.3.7 показана функция переходов этого ДКА.*

Таблица 1.3.7. Таблица переходов ДКА с минимальным количеством состояний.

	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Упражнение 1.3.3. *Пользуясь алгоритмом 1.3.5, постройте ДКА по следующим регулярным выражениям:*

а) $(a|b)^*$*;*

б) $(a^*|b^*)^*$*;*

в) $((\varepsilon|a)b^*)^*$*;*

г) $(a|b)^*abb(a|b)^*$*.*

Упражнение 1.3.4. *Можно доказать эквивалентность двух регулярных выражений, если показать, что их ДКА с минимальным количеством состояний одинаковы с точностью до имён состояний. Покажите таким образом, что регулярные выражения* $(a|b)^*$, $(a^*|b^*)^*$, *и* $((\varepsilon|a)b^*)^*$ — *эквивалентны.*

Упражнение 1.3.5. *Пользуясь алгоритмами 1.3.5 и 1.3.6, постройте ДКА с минимальным количеством состояний по следующим регулярным выражениям:*

а) $(a|b)^*a(a|b)$;

б) $(a|b)^*a(a|b)(a|b)$;

в) $(a|b)^*a(a|b)(a|b)(a|b)$;

г) $(a|b)^*a(a|b)(a|b)(a|b)(a|b)$.

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 $\{a^nb^n : n > 0\}$. 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 $xz, xyz, xyyz, \ldots, xy^nz, \ldots$, *принадлежат языку* L.

Покажем теперь, что язык $L = \{a^nb^n : n > 0\}$ — 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. Структура обрабатываемых файлов и порождаемый программный код

2.1. Структура файла с описанием лексического анализатора

Опишем структуру, которую должен иметь файл с описанием лексического анализатора.

Структура эта должна быть такой:

[%**scaner_name** *имя_сканера*]
[%**codes_type** *имя_типа_кодов_лексем*]
[%**ident_name** *имя_идентификатора*]
[%**token_fields** *добавляемые_поля_лексемы*]
[%**class_members** *добавляемые_члены_класса*]
[%**newline_is_lexem**]
%**codes**
имя_кода_лексемы{,*имя_кода_лексемы*}
[%**keywords** [*действия_по_завершении*:]
строка_ключевого_слова : *код_ключевого_слова*
{,*строка_ключевого_слова* : *код_ключевого_слова*}]
[%**idents**
{*описание_начала_идентификатора*}
{*описание_тела_идентификатора*}]
[%**delimiters** [*действия_по_завершении*:]
строка_разделителя : *код_разделителя*
{,*строка_разделителя* : *код_разделителя*}]
[%**numbers** [*действия_при_инициализации*]:[*действия_по_завершении*:]
{%**action** *имя_действия определение_действия* } {*выражение*}]
[%**strings** [*действия_при_инициализации*]:[*действия_по_завершении*:]
{%**action** *имя_действия определение_действия* } {*выражение*}]
[%**comments**
[%**single_lined** *начало_однострочного_комментария*]
[%**multilined** [%**nested**]
начало_многострочного_комментария : *конец_многострочного_комментария*]]

Прежде чем пояснить смысл каждой из приведённых только что конструкций, условимся, что

1) всё, заключённое в квадратные скобки зелёного цвета ([...]) является необязательным;
2) то, что заключено в фигурные скобки зелёного цвета ({...}), может повторяться любое число раз, в том числе и ни разу.

Прежде чем перейти к пояснению этой структуры, отметим, что под строковым литералом Мяуки (далее просто строковым литералом) будет пониматься любая (в том числе пустая) цепочка символов, заключённая в двойные кавычки. Если в этой последовательности нужно указать саму двойную кавычку, то эту кавычку нужно удвоить.

Перейдём теперь к пояснению структуры файла с описанием лексического анализатора (далее, для краткости, — сканера).

Прежде всего, если указана команда
%**scaner_name** *имя_сканера*
то в одном из заголовочных файлов появляется запись вида

```
class имя_сканера {
...
};
```

Сам же этот заголовочный файл будет называться `имя_сканера'.h`. Соответствующий файл реализации будет называться `имя_сканера'.cpp`, где `имя_сканера'` — `имя_сканера`, преобразованное к нижнему регистру. Принятое по умолчанию имя_сканера — `Scaner`.

Далее, если указана команда

%codes_type *имя_типа_кодов_лексем*

то раздел

%codes

имя_кода_лексемы{,*имя_кода_лексемы*}

порождает в файле `имя_сканера'.h` запись вида

```
enum имя_типа_кодов_лексем : unsigned short {
    NONE,
    UNKNOWN,
    имя_кода_лексемы1,
    ...
    имя_кода_лексемыN
};
```

где `имя_кода_лексемы1`, ..., `имя_кода_лексемыN` — имена кодов лексем, определённые в разделе **%codes**. Принятое по умолчанию имя типа кодов лексем — `Lexem_code`.

Команда

%ident_name *имя_идентификатора*

указывает имя кода лексемы для лексемы 'идентификатор'. Если в языке, для которого пишется сканер, идентификаторов нет, то команда **%ident_name** необязательна.

Если в описание лексемы нужно добавить какие–либо поля, то необходимо написать команду

%token_fields *добавляемые_поля_лексемы*

где *добавляемые_поля_лексемы* — строковый литерал с описанием нужных полей. Например, если лексема может принимать как значения типа `__float128`, так и значения типа `__int128`, причём поле типа `__float128` по условиям задачи нужно назвать `x`, а поле типа `__int128` — `y`, то строковый литерал с добавляемыми в лексему полями может выглядеть, например, так:

"__float128 x;
__int128 y;"

Кроме того, если в класс сканера требуется добавить члены, необходимые для каких–либо вычислений, то нужно написать

%class_members *добавляемые_члены_класса*

где под *добавляемые_члены_класса* понимается строковый литерал, содержащий перечень добавляемых в сканер членов. Если, например, нужно добавить

```
__int128 integer_value;
__float128 integer_part;
__float128 fractional_part;
__float128 exponent;
```

то вместо *добавляемые_члены_класса* нужно написать

"__int128 integer_value;
__float128 integer_part;
__float128 fractional_part;
__float128 exponent;"

Если необходимо, чтобы символ `'\n'` перехода на новую строку был отдельной лексемой, а не пробельным символом, то нужно указать команду **%newline_is_lexem**.

В обязательном разделе **%codes** содержится список разделённых запятыми идентификаторов (правила построения идентификаторов — такие же, что и в C++), представляющих собой имена кодов лексем.

Например, если имя перечисления с кодами лексемы не указано командой **%codes_type**, и раздел **%codes** имеет вид

%codes

Kw_if, Kw_then, Kw_else, Kw_endif

то будет порождено перечисление

```
enum Lexem_code : unsigned short {
    NONE,    UNKNOWN,
    Kw_if,   Kw_then,
    Kw_else, Kw_endif
};
```

Иными словами, всегда определяется два специальных кода лексем: `NONE`, обозначающее конец обрабатываемого текста, и `UNKNOWN`, обозначающее неизвестную лексему.

В разделе **%keywords** указываются ключевые слова языка, для которого пишется сканер, и соответствующие этим ключевым словам коды лексем, взятые из раздела **%codes**. Например, если имеются ключевые слова **if**, **then**, **else**, **endif**, и этим ключевым словам соответствуют коды лексем `Kw_if`, `Kw_then`, `Kw_else`, `Kw_endif`, то раздел **%keywords** должен иметь следующий вид:

```
%keywords
    ...
    "if" : Kw_if,
    "then" : Kw_then,
    "else" : Kw_else,
    "endif" : Kw_endif
    ...
```

Здесь многоточием обозначено (возможно, имеющееся) описание других ключевых слов.

В разделе **%idents** определяется структура идентификатора того языка, для которого пишется сканер. Более точно, *описание_начала_идентификатора* указывает, что может быть в начале идентификатора, а *описание_тела_идентификатора* — как устроено тело идентификатора.

В разделе **%delimiters** указываются разделители и знаки операций языка, для которого пишется сканер, и соответствующие этим разделителям и знакам операций коды лексем, взятые из раздела **%codes**. Например, если в языке есть разделители `<`, `>`, `<=`, `>=`, `=`, `!=`, с соответствующими кодами лексем `del_LT`, `del_GT`, `del_LEQ`, `del_GEQ`, `del_EQ`, `del_NEQ`, то раздел **%delimiters** должен иметь вид

```
%delimiters
    ...
    "<"  : del_LT,
    ">"  : del_GT,
    "<=" : del_LEQ,
    ">=" : del_GEQ,
    "="  : del_EQ,
    "!=" : del_NEQ
    ...
```

Здесь многоточием обозначено (возможно, имеющееся) описание других разделителей и знаков операций.

В разделе **%numbers** указывается регулярное выражение, определяющее числа, с внедрёнными в это регулярное выражение действиями. Каждое из действий должно быть описано командой
%action *имя_действия определение_действия*
где *имя_действия* — идентификатор языка C++, являющийся именем определяемого действия, а *определение_действия* — строковый литерал `myau`, содержащий код на C++, выполняющий нужное действие.

В разделе **%strings** описывается структура строковых и символьных литералов (если символьные литералы вообще есть) языка, для которого пишется сканер. Раздел **%strings** устроен так же, как и раздел **%numbers**. При этом при указании раздела **%strings** у класса сканера автоматически определяются члены `std::string buffer` и `int char_code`.

Наконец, в разделе **%comments** описывается структура комментариев языка, для которого пишется сканер.

Командой
%single_lined *начало_однострочного_комментария*
где *начало_однострочного_комментария* — строковый литерал, представляющий цепочку символов, являющуюся началом однострочного комментария, определяется структура однострочного комментария.

Командой же

%multilined [**%nested**]

начало_многострочного_комментария : *конец_многострочного_комментария*

определяется структура многострочного комментария.

А именно, *начало_многострочного_комментария* и *конец_многострочного_комментария* — строковые литералы, являющиеся цепочками символов, начинающих и заканчивающих многострочный комментарий. Если указано слово **%nested**, то многострочный комментарий может быть вложенным.

Поясним теперь, что в `Myau` понимается под началом идентификатора, концом идентификатора, и регулярным выражением:

описание_начала_идентификатора → *выр*

описание_тела_идентификатора → *выр*

выр → *выр*$_0$`{`|*выр*$_0$`}`

выр$_0$ → *выр*$_1$`{`*выр*$_1$`}`

выр$_1$ → символ | *класс_символов*

класс_символов → `[:Latin:]` | `[:latin:]` | `[:Russian:]` | `[:russian:]` | `[:bdigits:]` | `[:odigits:]` | `[:digits:]` | `[:xdigits:]` | `[:Letter:]` | `[:letter:]` | `[:nsq:]` | `[:ndq:]`

выражение → *выражение*$_0$ `{`|*выражение*$_0$ `}`

выражение$_0$ → *выражение*$_1$ `{`*выражение*$_1$`}`

выражение$_1$ → *выражение*$_2$`[`?|*|+`]`

выражение$_2$ → *выражение*$_3$`[`$*имя_действия*`]`

выражение$_3$ → символ | *класс_символов* | (*выражение*)

В этой грамматике под словом „символ“ понимается следующее. Любой непробельный символ, кроме символов `'|'`, `'*'`, `'+'`, `'?'`, `'$'`, `'\'`, `'"'`, и символа перехода на новую строку, в файле с описанием сканера представляет самого себя. Если же эти символы нужно указать в регулярном выражении, то следует их записывать как `'\|'`, `'\*'`, `'\+'`, `'\?'`, `'\$'`, `'\\'`, `'\"'`, `'\n'` соответственно. При этом все пробельные символы (то есть символы, коды которых не превосходят кода пробела) генератором лексических анализаторов Мяука игнорируются.

В приводимой ниже таблице поясняются допускаемые классы символов.

Таблица 2.1.1. Допускаемые классы символов.

Класс символов	Пояснение
`[:Latin:]`	Прописные латинские буквы от 'A' до 'Z'.
`[:latin:]`	Строчные латинские буквы от 'a' до 'z'.
`[:Russian:]`	Прописные русские буквы от 'А' до 'Я' (включая букву 'Ё').
`[:russian:]`	Строчные русские буквы от 'а' до 'я' (включая букву 'ё').
`[:bdigits:]`	Символы двоичных цифр, т.е. символы '0' и '1'.
`[:odigits:]`	Символы восьмеричных цифр, т.е. символы '0', '1', '2', '3', '4', '5', '6', '7'.
`[:digits:]`	Символы десятичных цифр, т.е. символы '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.
`[:xdigits:]`	Символы шестнадцатеричных цифр, т.е. символы '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f'.
`[:Letter:]`	Прописные латинские буквы от 'A' до 'Z' и прописные русские буквы от 'А' до 'Я' (включая букву 'Ё').
`[:letter:]`	Строчные латинские буквы от 'a' до 'z' и строчные русские буквы от 'а' до 'я' (включая букву 'ё').
`[:nsq:]`	Символы, отличные от одинарной кавычки (').
`[:ndq:]`	Символы, отличные от двойной кавычки (").

Из всех этих классов символов классы `[:nsq:]` и `[:ndq:]` допускаются только в разделе **%strings**.

2.2. Порождаемый программный код

Прежде чем описать порождаемый программный код, условимся, что все порождаемые файлы будут в кодировке UTF-8.

Далее, порождаемому коду сканера потребуется подсчёт ошибок. Поэтому потребуется класс для подсчёта ошибок. Заголовочный файл будет называться `error_count.h`, а файл реализации — `error_count.cpp`. Содержимое этих файлов приводится в следующих двух листингах.

```
/* Файл error_count.h. */
#ifndef ERROR_COUNT_H
#define ERROR_COUNT_H
  class Error_count {
    public:
      int get_number_of_errors();
      void increment_number_of_errors();
      void print_number_of_errors();
      Error_count() : number_of_errors(0) {};
    private:
      int number_of_errors;
  };
#endif
```

```
/* Файл error_count.cpp. */
#include "error_count.h"
#include <cstdio>

int Error_count::get_number_of_errors(){
  return number_of_errors;
}

void Error_count::increment_number_of_errors(){
  ++number_of_errors;
}

void Error_count::print_number_of_errors(){
  printf ("Всего ошибок: %d\n", number_of_errors);
}
```

Далее, потребуется функция определения размера файла. Заголовочный файл назовём `fsize.h`, а файл реализации — `fsize.cpp`. Содержимое этих файлов указано в двух следующих листингах.

```
/* Файл fsize.h. */
#ifndef FSIZE_H
#define FSIZE_H
#include <cstdio>
/* Данная функция выдаёт размер файла в байтах, если
* fptr != NULL, и (-1) в противном случае. */
long fsize(FILE* fptr);
#endif
```

```
/* Файл fsize.cpp. */
#include "../include/fsize.h"
#include <cstdio>
long fsize(FILE* fptr){
    long ret_val = -1;
    if(fptr){
        long current_pos = ftell(fptr);
        fseek(fptr, 0L, SEEK_END);
        ret_val = ftell(fptr);
        fseek(fptr, current_pos, SEEK_SET);
    }
    return ret_val;
}
```

Вся внутренняя обработка текста будет вестись для кодировки UTF32, при этом каждый символ будет иметь тип char32_t. При этом считаем, что файлы с текстом, обрабатываемые сгенерированным сканером, будут в кодировке UTF-8. В связи с этим потребуется, во–первых, функция, читающая всё содержимое файла, и, во–вторых, функция, преобразующая строки и символы из кодировки UTF32 в кодировку UTF-8 и обратно. Ниже приведены соответстующие заголовочные файлы и файлы реализации.

```
/* Файл char_conv.h.*/
/**
\file

\brief Заголовочный файл с прототипами функций, преобразующих строки из кодировки
UTF-8 в кодировку UTF-32 и наоборот.
*/

#ifndef CHAR_CONV_H
#define CHAR_CONV_H

#include <string>

/**
\function utf8_to_u32string
 Данная функция по строке в кодировке UTF-8 строит строку в кодировке UTF-32.

\param  utf8str - строка в кодировке UTF-8 с завершающим нулевым символом

\return значение типа std::u32string, представляющее собой ту же строку,
 но в кодировке UTF-32
*/
std::u32string utf8_to_u32string(const char* utf8str);

/**
\function u32string_to_utf8
 Данная функция по строке в кодировке UTF-32 строит строку в кодировке UTF-8.

\param [in] u32str - строка в кодировке UTF-32

\return значение типа std::string, представляющее собой ту же строку,
 но в кодировке UTF-8
*/
std::string u32string_to_utf8(const std::u32string& u32str);

/**
\function char32_to_utf8
По символу в кодировке UTF-32 строит строку, состоящую из байтов, представляющих
тот же символ, но в кодировке UTF-8.

\param [in] c - символ в кодировке UTF-32

\return Значение типа std::string, состоящее из байтов, представляющих
тот же символ, но в кодировке UTF-8.
*/
std::string char32_to_utf8(const char32_t c);
#endif
```

```
/* Файл file_contents.h. */
#ifndef FILE_CONTENTS_H
#define FILE_CONTENTS_H
```

```
#include <string>
#include <utility>

/** Коды возврата из функции get_contents. */
enum class Get_contents_return_code{
    Normal,           ///< Этот код означает, что всё прошло успешно.
    Impossible_open,  ///< Этот код означает, что не удалось открыть файл.
    Read_error        ///< Этот код означает, что во время чтения файла произошла ошибка.
};

using Contents  = std::pair<Get_contents_return_code, std::string>;

/**
   Возвращает всё содержимое файла с заданным именем.
   \param [in] name --- имя читаемого файла
   \returns Пару из кода возврата (первая компонента) и значения, имеющего
тип std::string (вторая компонента). При возникновении ошибки вторая компонента
возвращаемого значения представляет собой пустую строку.
*/
Contents get_contents(const char* name);
#endif

/* Файл char_conv.cpp.  */
#include "../include/char_conv.h"

std::string char32_to_utf8(const char32_t c){
    std::string s;
    char c1, c2, c3, c4;
    char32_t temp = c;
    switch(c){
        case 0x0000'0000 ... 0x0000'007f:
            s += static_cast<char>(c);
            break;

        case 0x0000'0080 ... 0x0000'07ff:
            c1 = 0b110'0'0000 | (temp >> 6);
            c2 = 0b10'00'0000 | (temp & 0b111'111);
            s += c1; s += c2;
            break;

        case 0x0000'0800 ... 0x0000'ffff:
            c3 = 0b10'00'0000 | (temp & 0b111'111);
            temp >>= 6;
            c2 = 0b10'00'0000 | (temp & 0b111'111);
            temp >>= 6;
            c1 = 0b1110'0000 | c;
            s += c1; s += c2; s += c3;
            break;

        case 0x0001'0000 ... 0x001f'ffff:
            c4 = 0b10'00'0000 | (temp & 0b111'111);
            temp >>= 6;
            c3 = 0b10'00'0000 | (temp & 0b111'111);
            temp >>= 6;
            c2 = 0b10'00'0000 | (temp & 0b111'111);
            temp >>= 6;
            c1 = 0b11110'000 | c;
```

```
            s += c1; s += c2; s += c3; s += c4;
            break;

        default:
            ;
    }
    return s;
}

std::string u32string_to_utf8(const std::u32string& u32str){
    std::string s;
    for(const char32_t c : u32str){
        s += char32_to_utf8(c);
    }
    return s;
}

std::u32string utf8_to_u32string(const char* utf8str){
    std::u32string s;
    enum class State{
        Start_state,                Three_byte_char_second_byte,
        Four_byte_char_second_byte, Four_byte_char_third_byte,
        Last_byte_of_char
    };
    State state = State::Start_state;
    char32_t current_char = 0;
    while(char c = *utf8str++){
        switch(state){
            case State::Start_state:
                if(c >= 0){
                    s += c;
                }else if((c & 0b1110'0000) == 0b1100'0000){
                    current_char = c & 0b0001'1111;
                    state = State::Last_byte_of_char;
                }else if((c & 0b1111'0000) == 0b1110'0000){
                    current_char = c & 0b0000'1111;
                    state = State::Three_byte_char_second_byte;
                }else if((c & 0b1111'1000) == 0b1111'0000){
                    current_char = c & 0b0000'0111;
                    state = State::Four_byte_char_second_byte;
                }
                break;

            case State::Last_byte_of_char:
                current_char = (current_char << 6) | (c & 0b0011'1111);
                s += current_char;
                state = State::Start_state;
                break;

            case State::Three_byte_char_second_byte:
                current_char = (current_char << 6) | (c & 0b0011'1111);
                state = State::Last_byte_of_char;
                break;

            case State::Four_byte_char_second_byte:
                current_char = (current_char << 6) | (c & 0b0011'1111);
```

```
                state = State::Four_byte_char_third_byte;
                break;

            case State::Four_byte_char_third_byte:
                current_char = (current_char << 6) | (c & 0b0011'1111);
                state = State::Last_byte_of_char;
                break;
        }
    }
    return s;
}

/* Файл file_contents.cpp. */
#include "../include/file_contents.h"
#include "../include/fsize.h"
#include <cstdio>
#include <memory>

Contents get_contents(const char* name){
    Contents result = std::make_pair(Get_contents_return_code::Normal, "");
    FILE* fptr = fopen(name, "rb");
    if(!fptr){
        result.first = Get_contents_return_code::Impossible_open;
        return result;
    }
    long file_size = fsize(fptr);
    if(!file_size){
        return result;
    }
    auto   test_text = std::make_unique<char[]>(file_size + 1);
    char*  q         = test_text.get();
    size_t fr        = fread(q, 1, file_size, fptr);
    if(fr < (unsigned long)file_size){
        fclose(fptr);
        result.first = Get_contents_return_code::Read_error;
        return result;
    }
    test_text[file_size] = 0;
    fclose(fptr);
    result.second = std::string(test_text.get());
    return result;
}
```

Далее, в сгенерированных файлах используется проверка принадлежности неотрицательного целого числа, находящегося в диапазоне от 0 до 63, множеству таких чисел. Для этого нужен приводимый ниже файл `belongs.h`.

```
#ifndef BELONGS_H
#define BELONGS_H
/* Данная функция проверяет, принадлежит ли элемент element множеству s. При этом
 * считаем, что множество s состоит из не более чем 64 элементов, так что в качестве
 * внутреннего представления множества используется тип uint64_t. Если бит с номером
 * i внутреннего представления равен 1, то элемент i принадлежит множеству,
 * иначе --- не принадлежит. */
inline uint64_t belongs(uint64_t element, uint64_t s){
    return s & (1ULL << element);
}
#endif
```

Для случая проверки принадлежности числа большему множеству нужен файл `operations_with_sets.h`:

```
#ifndef OPERATIONS_WITH_SETS_H
#define OPERATIONS_WITH_SETS_H

#include <set>
#include <cstdio>
/**
    В данном файле определяются теоретико--множественные операции
    со стандартными контейнерами std::set.
*/
namespace operations_with_sets{
    //! Функция single_elem возвращает множество, состоящее из одного элемента.
    template<typename T>
    std::set<T> single_elem(const T& x){
        std::set<T> s;
        s.insert(x);
        return s;
    }

    /** Функция печати элементов множества. Принимает в качестве
        аргумента функцию печати элемента множества. */
    template<typename T>
    void print_set(const std::set<T>& a, void (*print_elem)(const T&)){
        if(a.empty()){
            printf("{}");
            return;
        }
        auto first       = a.begin();
        auto before_last = --a.end();
        putchar('{');
        for(auto i = first; i != before_last; ++i){
            print_elem(*i);
            putchar(',');
        }
        print_elem(*before_last);
        putchar('}');
    }

    /** Проверка принадлежности элемента x множеству a. Если элемент x
        множеству a принадлежит, то возвращается true, иначе ---
        возвращается false. */
    template<typename T>
    bool is_elem(const T& x, const std::set<T>& a){
        return a.find(x) != a.end();
    }

    /** Объединение множеств a и b, то есть множество, содержащее и
        элементы множества a, и элементы множества b. */
    template<typename T>
    std::set<T> operator + (const std::set<T>& a, const std::set<T>& b){
        std::set<T> s = a;
        s.insert(b.begin(), b.end());
        return s;
    }
```

```
    /** Теоретико--множественная разность множеств a и b (обозначается в
        теории множеств как a \ b), то есть множество, состоящее лишь из тех
        элементов множества a, которые не принадлежат множеству b. */
    template<typename T>
    std::set<T> operator - (const std::set<T>& a, const std::set<T>& b){
        std::set<T> s = a;
        for(const auto x : b){
            s.erase(x);
        }
        return s;
    }

    /** Пересечение множеств a и b, то есть множество, состоящее в точности из
        тех элементов, которые принадлежат и a, и b. */
    template<typename T>
    std::set<T> operator * (const std::set<T>& a, const std::set<T>& b){
        std::set<T> s;
        for(const auto& x : a){
            if(is_elem(x, b)){
                s.insert(x);
            }
        }
        return s;
    }

    /** Симметрическая разность множеств a и b, то есть объединение этих
        множеств с выкинутыми общими элементами. */
    template<typename T>
    std::set<T> operator ^ (const std::set<T>& a, const std::set<T>& b){
        return (a - b) + (b - a);
    }

    /** Проверяет, является ли множество a подмножеством множества b,
        возможно, совпадающим с b. */
    template<typename T>
    bool is_subseteq(const std::set<T>& a, const std::set<T>& b){
        std::set<T> s = (a * b) ^ a;
        return s.empty();
    }
};
#endif
```

Далее, для облегчения реализации таблиц символов строковые представления идентификаторов и строковых литералов хранятся в специальных таблицах, и в лексеме возвращается не строковое представление идентификатора или строкового литерала, а его индекс в таблице.

При этом если в языке, для которого генерируется лексический анализатор (далее называемый сканером) имеются идентификаторы, то в структуру со сведениями о лексеме добавляется поле `size_t ident_index`, содержащее индекс идентификатора в таблице идентификаторов. Если же в языке, для которого генерируется сканер, имеются строковые литералы, то в структуру со сведениями о лексеме добавляются поле `size_t string_index`, содержащее индекс строкового литерала в таблице строковых литералов, и поле `char32_t c`, содержащее символьный литерал.

Таблицы идентификаторов и строковых литералов представляют собой префиксные деревья (английское название — trie).

Под префиксным деревом здесь понимается структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки[1].

[1] В данном случае строки не обязаны состоять из символов в смысле какой–либо кодировки (кодировки DOS, Windows, и т.д.).

Нагруженное дерево отличается от обычных n–арных деревьев тем, что в его узлах хранятся не ключи, а односимвольные метки. Ключом же, соответствующим некоторому узлу, является путь от корня дерева до этого узла, а точнее, строка составленная из меток узлов, повстречавшихся на этом пути. В таком случае корень дерева, очевидно, соответствует пустому ключу. Таким образом, потомки узла имеют общий префикс, откуда и произошло название данной структуры данных. Иные названия на русском языке — бор [8], луч [9], и нагруженное дерево [10, с.152]. Пример префиксного дерева (дерево для идентификаторов `get`, `her`, `him`, `his`):

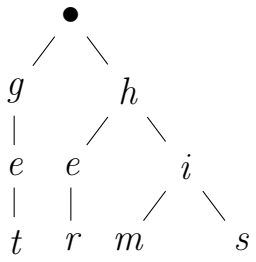


Рис. 2.2.1. Префиксное дерево для идентификаторов `get`, `her`, `him`, `his`.

Внутреннее представление будет таким:

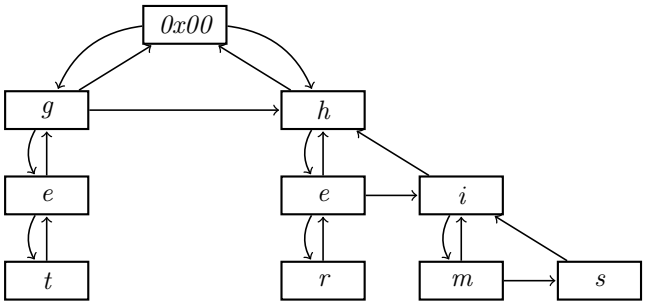


Рис. 2.2.2. Внутреннее представление префиксного дерева для идентификаторов `get`, `her`, `him`, `his`.

Поясним рисунок 2.2.2. Каждый узел помечен символом типа `char32_t`. А именно, корень помечен символом с кодом ноль. Потомки корня помечены первыми символами вставленных строк. Если узел не является корнем, то его потомки помечаются символами, которые во вставленных строках идут сразу же после префикса, получающегося проходом от корня до этого узла. Например, потомки крайнего правого узла на третьем уровне изображённого на рис. 2.2.2 дерева (этот узел помечен символом i) помечены символами m и s. Для этого узла префикс, получающийся проходом от корня, представляет собой строку hi, являющуюся префиксом идентификаторов `him` и `his`.

У каждого узла, изображённого на рис. 2.2.2, помимо поля, содержащего символ, который помечает узел, имеются также поля–указатели. А именно, имеются следующие поля: указатель на узел–предок, указатель на следующий потомок узла–предка, и указатель на список узлов–потомков. Все узлы–потомки организованы в виде односвязного списка. При этом под указателем на список узлов–потомков понимается указатель на первый элемент списка.

Чтобы явно не выделять память, узлы будем хранить в стандартном контейнере `vector`. Новые узлы будем добавлять в конец контейнера. Поскольку все узлы хранятся в контейнере `vector`, то поля–указатели заменить полями–индексами. Иными словами, каждый узел реализован в виде структуры, хранящей следующие данные:

1) символ, которым помечен узел;

2) целочисленное значение, хранящее индекс узла–предка в контейнере `vector`;

3) целочисленное значение, хранящее индекс в контейнере `vector` первого элемента в односвязном списке узлов–потомков;

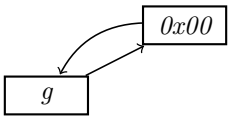
4) целочисленное значение, хранящее индекс в контейнере `vector` следующего потомка узла–предка.

Реализация пустого префиксного дерева такова: контейнер `vector`, содержащий узлы, содержит единственный узел (корень дерева), помеченный символом с кодом 0, и все поля–индексы которого равны нулю.

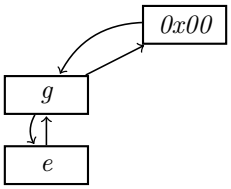
Далее, синтаксическому анализатору нужно выводить идентификаторы при выводе сообщений об ошибках (например, при обнаружении несовместимости типов и при обнаружении идентификатора, который используется, но не описан) по известным индексам идентификаторов в таблице идентификаторов. Для упрощения вывода идентификаторов на экран добавим к внутреннему представлению узла поле, равное длине пути от корня до данного узла.

Поясним, как осуществляется вставка в префиксное дерево. Рассмотрим сначала случай вставки в пустое дерево. Вставлять будем идентификатор `get`. Итак, до вставки дерево выглядело так:

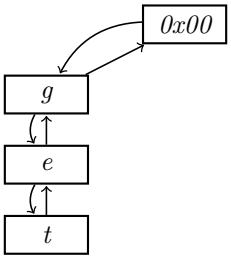
При вставке идентификатора `get` нужно последовательно пройти по всем символам этого идентификатора. Первый символ идентификатора `get` — символ '`g`'. Поскольку дерево пусто, то оно состоит только из корня, и, следовательно, корень дерева не имеет потомков. Поэтому в список потомков корня нужно вставить узел, помеченный первым символом вставляемого идентификатора, то есть символом '`g`'. После этого дерево примет вид



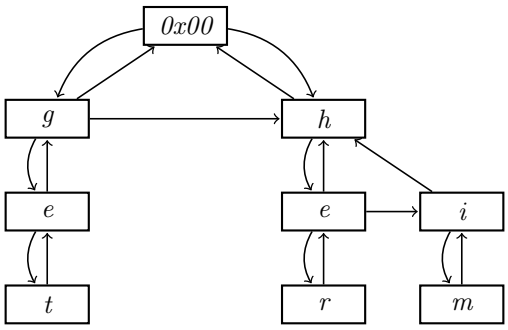
Теперь считаем вставленный узел корнем поддерева, и в это поддерево вставляем оставшуюся часть идентификатора, то есть строку `et`. Первым символом этой строки является символ '`e`'. Поскольку корень рассматриваемого поддерева не имеет потомков, то в список потомков корня рассматриваемого поддерева нужно вставить узел, помеченный первым символом вставляемой строки `et`, то есть символом '`e`'. После этого дерево примет вид



Считаем вставленный узел корнем поддерева, и в это поддерево вставляем оставшуюся часть идентификатора, то есть строку `t`. Первым (и единственным) символом данной строки является символ '`t`'. Поскольку корень рассматриваемого поддерева не имеет потомков, то в список потомков корня рассматриваемого поддерева нужно вставить узел, помеченный первым символом вставляемой строки `t`, то есть символом '`t`'. Итак, после вставки идентификатора `get` в пустое дерево получим следующее префиксное дерево:



Рассмотрим теперь следующее префиксное дерево, содержащее идентификаторы `get`, `her`, `him`:



Вставим в это дерево идентификатор `his`. Первым символом вставляемого идентификатора является символ '`h`'. Поскольку у корня префиксного дерева есть потомок, помеченный указанным символом, то вставляем оставшуюся часть идентификатора, строку `is`, в поддерево, корнем которого является потомок корня дерева, помеченный символом '`h`'. Среди потомков корня рассматриваемого поддерева есть узел, помеченный первым символом строки `is`, то есть символом '`i`'. Поэтому в поддерево, корнем которого является помеченный символом '`i`' узел, нужно вставить оставшуюся часть идентификатора `his`, строку `s`. У корня того поддерева, в котором находимся в настоящий момент, имеется потомок, помеченный символом '`m`', но нет потомка, помеченного символом '`s`'. Следовательно, нужно в список потомков вставить корня текущего поддерева вставить узел, помеченный символом '`s`'. В результате получим дерево, изображённое на рис. 2.2.2.

Класс префиксного дерева для строковых литералов и для идентификаторов будет наследником класса префиксного дерева, параметризованного типом хранимых в узлах данных. Параметризованный класс префиксного дерева будет находиться в файле `trie.h`, содержимое которого приводится ниже.

```
/* Файл trie.h. */
#ifndef TRIE_H
#define TRIE_H

#include <vector>
#include <map>
#include <utility>
#include <algorithm>
#include <string>
#include <set>

template<typename T>
class Trie {
public:
    /* Конструктор по умолчанию. */
    Trie<T>();
    /* Деструктор. */
    ~Trie() = default;
    /* Копирующий конструктор. */
    Trie(const Trie<T>& orig) = default;
    /* Функция вставки в префиксное дерево. */
    size_t insert(const std::basic_string<T>& s);
    /* Функция, вычисляющая максимальную степень вершин префиксного
     * дерева (корень дерева не учитывается). */
    size_t maximal_degree();
protected:
    /* тип узла префиксного дерева: */
    struct node{
      size_t parent, first_child, next;
      /* Все узлы-потомки текущего узла организованы в виде односвязного списка, первым
       * элементом которого является элемент с индексом first_child. В поле parent
```

```
        * содержится индекс родительского узла, а в поле next -- следующего потомка
        * родительского узла. Если у текущего узла потомков нет, то в поле first_child
        * содержится нуль. Аналогично, последний элемент в списке потомков в поле next
        * содержит нуль. Здесь под индексом понимается индекс в поле node_buffer,
        * представляющем собой вектор (в смысле библиотеки STL) из узлов префиксного
        * дерева. */
       size_t path_len; /* в этом поле содержится длина пути
                         * от текущего узла до корня дерева */
       size_t degree; /* В этом поле содержится степень узла,
                       * то есть количество выходящих из узла рёбер. */
       T c; /* в этом поле содержится символ вставленной строки,
             * являющийся меткой текущего узла. */
       node(){
         next = parent = path_len = first_child = 0;
         degree = 0; c = T();
       }
     };
     std::vector<node>   node_buffer;
     std::vector<size_t> nodes_indeces;
     /* Функция, добавляющая к списку потомков узла parent_idx узел, помеченный
      * значением x типа T. Функция возвращает индекс вставленного узла. */
     size_t add_child(size_t parent_idx, T x);
     /* Эта функция выполняет (возможно, необходимые) действия
      * по окончании вставки последнего символа. */
     virtual void post_action(const std::basic_string<T>& s, size_t n){ };
};

template<typename T>
Trie<T>::Trie(){
    node_buffer = std::vector<node>(1);
    nodes_indeces = std::vector<size_t>();
}

template<typename T>
size_t Trie<T>::maximal_degree(){
    size_t deg = 0;
    size_t len = node_buffer.size();
    for(int i = 1; i < len; i++){
        deg = std::max(deg,node_buffer[i].degree);
    }
    return deg;
}

template<typename T>
size_t Trie<T>::add_child(size_t parent_idx, T x){
    size_t current, previous;
    node   temp;
    current = previous = node_buffer[parent_idx].first_child;
    /* В переменной temp содержится узел, который, возможно, придётся вставить. */
    temp.c = x; temp.degree = 0;
    temp.next = 0; temp.parent = parent_idx;
    temp.path_len = node_buffer[parent_idx].path_len + 1;
    if(!current){
        /* Здесь можем оказаться, лишь если у узла с индексом parent_idx потомков
         * вообще нет. Значит добавляемый узел будет первым в списке потомков. При
         * этом степень узла parent_idx увеличится на единицу, и станет равна 1. */
        node_buffer.push_back(temp);
```

```
        size_t child_idx = node_buffer.size() - 1;
        node_buffer[parent_idx].first_child = child_idx;
        node_buffer[parent_idx].degree = 1;
        return child_idx;
    }
    while(current){
        // Если же потомки есть, то нужно пройти по списку потомков.
        node current_node = node_buffer[current];
        if(current_node.c == x){
          /* Если потомок, помеченный нужным символом (символом x),
           * есть, то нужно вернуть индекс этого потомка. */
          return current;
        }else{
          previous = current; current = current_node.next;
        }
    }
    /* Если же такого потомка нет, то нужно этого потомка добавить
     * в конец списка потомков.*/
    node_buffer.push_back(temp);
    size_t next_child = node_buffer.size() - 1;
    node_buffer[previous].next = next_child;
    node_buffer[parent_idx].degree++;
    return next_child;
}

template<typename T>
size_t Trie<T>::insert(const std::basic_string<T>& s){
    ssize_t len = s.length();
    size_t current_root = 0;
    for (ssize_t i = 0; i < len; i++) {
        current_root = add_child(current_root,s[i]);
    }
    nodes_indeces.push_back(current_root);
    post_action(s,current_root);
    return current_root;
}
#endif
```

Описание префиксного дерева для идентификаторов и строковых литералов будет храниться в файле `char_trie.h`, а реализация — в файле `char_trie.cpp`. Содержимое этих файлов приведено ниже.

```
/* Файл char_trie.h. */
#ifndef CHAR_TRIE_H
#define CHAR_TRIE_H

#include "../include/trie.h"

struct Char_trie_as_map {
    std::map<size_t,char32_t*> *m;
    ~Char_trie_as_map();
};

class Char_trie : public Trie<char32_t>{
public:
    virtual ~Char_trie() { };
    /* Конструктор по умолчанию. */
    Char_trie(){};
```

```
    /* Копирующий конструктор. */
    Char_trie(const Char_trie& orig) = default;
    /* Функция, по индексу idx строящая строку в стиле C,
     * соответствующую индексу idx. */
    char32_t* get_cstring(size_t idx);
    /* Функция, по индексу idx строящая строку типа u32string,
       соответствующую индексу idx. */
    std::u32string get_string(size_t idx);
    /* Функция, возвращающая префиксное дерево в виде
     * отображения индексов строк в строки в стиле C. */
    Char_trie_as_map as_map();
    /* Функция вывода строки, которой соответствует индекс idx, на экран. */
    void print(size_t idx);
    /* Следующая функция по индексу строки возвращает длину этой строки. */
    size_t get_length(size_t idx);
};

#endif

/* Файл char_trie.cpp. */
#include "../include/char_conv.h"
#include "../include/char_trie.h"
#include <vector>
#include <map>
#include <utility>
#include <algorithm>
#include <string>
#include <set>

Char_trie_as_map::~Char_trie_as_map(){
    for(auto x : *m){
        delete [] x.second;
    }
}

Char_trie_as_map Char_trie::as_map(){
    Char_trie_as_map t;
    t.m = new std::map<size_t,char32_t*>();
    for(auto x : nodes_indeces){
        t.m -> insert({x,get_cstring(x)});
    }
    return t;
}

char32_t* Char_trie::get_cstring(size_t idx){
    size_t id_len = node_buffer[idx].path_len;
    char32_t* p = new char32_t[id_len + 1];
    p[id_len] = 0;
    size_t current = idx;
    size_t i       = id_len-1;
    /* Поскольку idx -- индекс элемента в node_buffer, содержащего последний символ
     * вставленной строки, а каждый элемент вектора node_buffer содержит поле parent,
     * указывающее на элемент с предыдущим символом строки, то для получения
     * вставленной строки, которой соответствует индекс idx, в виде массива символов,
     * нужно пройтись от элемента с индексом idx к корню. При этом символы вставленной
     * строки будут читаться от её конца к началу. */
    for( ; current; current = node_buffer[current].parent){
```

```
        p[i--] = node_buffer[current].c;
    }
    return p;
}

std::u32string Char_trie::get_string(size_t idx){
    char32_t* p = get_cstring(idx);
    std::u32string s = std::u32string(p);
    delete [] p;
    return s;
}

void Char_trie::print(size_t idx){
    std::u32string s32 = get_string(idx);
    std::string    s8  = u32string_to_utf8(s32);
    printf("%s",s8.c_str());
}

size_t Char_trie::get_length(size_t idx){
    return node_buffer[idx].path_len;
}
```

Кроме того, генерируется файл `errors_and_tries.h`:

```
/* Файл errors_and_tries.h. */
ifndef ERRORS_AND_TRIES_H
#define ERRORS_AND_TRIES_H

#include "../include/error_count.h"
#include "../include/char_trie.h"
#include <memory>
struct Errors_and_tries{
    std::shared_ptr<Error_count> ec;
    std::shared_ptr<Char_trie>   ids_trie;
    std::shared_ptr<Char_trie>   strs_trie;

    Errors_and_tries()  = default;
    ~Errors_and_tries() = default;
};
#endif
```

Для порождения реализации автоматов, распознающих ряд видов лексем, нужна функция, ищущая символ в заканчивающейся символом с кодом ноль строке, состоящей из символов типа `char32_t`. Прототип этой функции содержится в файле `search_char.h`, а реализация — в файле `search_char.cpp`. Ниже приведено содержимое каждого из этих файлов.

```
/* Файл search_char.h. */
#define THERE_IS_NO_CHAR (-1)
/**
 * \function search_char
 * Данная функция ищет заданный символ типа char32_t в строке,
 * состоящей из символов такого типа и завершающейся нулевым
 * символом.
 *
 * \param [in] c --- искомый символ
 * \param [in] array --- строка в которой ищется символ
 * \return смещение (в символах) от начала строки, если
 * искомый символ в строке есть, и (-1) в противном случае
```

```
 */
int search_char(char32_t c, const char32_t* array);
#endif

/* Файл search_char.cpp. */
#include "../include/search_char.h"
int search_char(char32_t c, const char32_t* array){
    char32_t ch;
    int      curr_pos = 0;
    for(char32_t* p = const_cast<char32_t*>(array); (ch = *p++); ){
        if(ch == c){
            return curr_pos;
        }
        curr_pos++;
    }
    return THERE_IS_NO_CHAR;
}
```

Для инициализации ряда порождаемых автоматов потребуется функция, определяющая состояние, с которого автомат начнёт работу. Прототип функции будет содержаться в файле `get_init_state.h`, а реализация — в файле `get_init_state.cpp`. Приведём содержимое этих файлов.

```
/* Файл get_init_state.h. */
#ifndef GET_INIT_STATE_H
#define GET_INIT_STATE_H

struct State_for_char{
    unsigned st;
    char32_t c;
};

/* Функция get_init_state инициализирует конечный автомат. Делает она это так: ищет
 * символ sym в таблице sts, состоящей из пар (состояние, символ) и имеющей размер
 * n, двоичным поиском по второму компоненту пары. После нахождения выдаётся
 * первая компонента пары. В качестве алгоритма двоичного поиска используется
 * алгоритм B из раздела 6.2.1 монографии "Кнут Д.Э. Искусство программирования.
 * Т.3. Сортировка и поиск. 2-е изд.: Пер. с англ. --- М.: Вильямс, 2008.". При
 * этом в нашем случае не может быть, чтобы нужный элемент в таблице sts
 * отсутствовал. */
int get_init_state(char32_t sym, const State_for_char sts[], int n);
#endif

/* Файл get_init_state.cpp. */
#include "../include/get_init_state.h"
int get_init_state(char32_t sym, const State_for_char sts[], int n){
    int lower, upper, middle;
    lower = 0; upper = n - 1;
    while(lower <= upper){
        middle = (lower + upper) >> 1;
        char32_t c_ = sts[middle].c;
        if(sym == c_){
            return sts[middle].st;
        }else if(sym > c_){
            lower = middle + 1;
        }else{
            upper = middle - 1;
        }
    }
```

```
    return -1;
}
```

Сканеру нужно будет передавать обрабатываемый текст и указатель на текущее местоположение в этом тексте. Кроме того, бывает, что для разных кусков обрабатываемого текста требуются разные сканеры, и эти сканеры должны разделять сведения об обрабатываемом тексте и текущем в нём положении. Поэтому генерируется файл `location.h`, содержащий такие сведения:

```
/* Файл location.h. */
#ifndef LOCATION_H
#define LOCATION_H

#include <memory>
/* Следующая структура описывает текущее положение в обрабатываемом тексте.
 * В конструктор сканера нужно передавать умный указатель на
 * разделяемые сведения о текущем местоположении. */

//#include <cstddef>
struct Location {
    char32_t* pcurrent_char; /* указатель на текущий символ */
    size_t    current_line; /* номер текущей строки обрабатываемого текста */

    Location() : pcurrent_char(nullptr), current_line(1) {};
    Location(char32_t* txt) : pcurrent_char(txt), current_line(1) {};
};

using Location_ptr = std::shared_ptr<Location>;
#endif
```

Наконец, генерируемый сканер будет наследником шаблонного класса сканера. Шаблонны класс сканера содержится в файле `abstract_scaner.h`:

```
/* Файл abstract_scaner.h. */
#ifndef ABSTRACT_SCANER_H
#define ABSTRACT_SCANER_H

#include <string>
#include <memory>
#include "../include/error_count.h"
#include "../include/char_trie.h"
#include "../include/location.h"
#include "../include/errors_and_tries.h"

template<typename Lexem_type>
class Abstract_scaner{
public:
    Abstract_scaner<Lexem_type>() = default;
    Abstract_scaner(Location_ptr location, const Errors_and_tries& et);
    Abstract_scaner(const Abstract_scaner<Lexem_type>& orig) = default;
    virtual ~Abstract_scaner() = default;
    /* Функция back() возвращает текущую лексему во входной поток.*/
    void back();
    /* Функция current_lexem() возвращает сведения о текущей
     * лексеме (код лексемы и значение лексемы). */
    virtual Lexem_type current_lexem() = 0;
    /* Функция lexem_begin_line_number() возвращает номер строки
     * обрабатываемого текста, с которой начинается лексема,
     * сведения о которой возвращены функцией current_lexem(). */
```

```
    size_t lexem_begin_line_number();
protected:
    int                         state; /* текущее состояние текущего автомата */

    Location_ptr                loc;
    char32_t*                   lexem_begin; /* указатель на начало лексемы */
    char32_t                    ch;          /* текущий символ */

    /* множество категорий символов, которым принадлежит
     * текущий символ */
    uint64_t                    char_categories;

    /* промежуточное значение сведений о лексеме */
    Lexem_type                  token;

    /* номер строки, с которой начинается текущая лексема */
    size_t                      lexem_begin_line;

    /* указатель на класс, подсчитывающий количество ошибок: */
    std::shared_ptr<Error_count> en;
    /* указатель на префиксное дерево для идентификаторов: */
    std::shared_ptr<Char_trie>   ids;
    /* указатель на префиксное дерево для строк: */
    std::shared_ptr<Char_trie>   strs;

    /*буфер для записи обрабатываемого идентификатора или строки: */
    std::u32string              buffer;
};

template<typename Lexem_type>
Abstract_scaner<Lexem_type>::Abstract_scaner(Location_ptr location,
                                             const Errors_and_tries& et){
    ids = et.ids_trie; strs = et.strs_trie; en = et.ec;
    loc = location;
    lexem_begin = location->pcurrent_char;
    lexem_begin_line = 1;
}

template<typename Lexem_type>
void Abstract_scaner<Lexem_type>::back(){
    loc->pcurrent_char = lexem_begin;
    loc->current_line =  lexem_begin_line;
}

template<typename Lexem_type>
size_t Abstract_scaner<Lexem_type>::lexem_begin_line_number(){
    return lexem_begin_line;
}
#endif
```

Глава 3. Пример программы тестирования работы сгенерированного сканера

Наконец, приведём пример программы, тестирующей работу сгенерированного сканера.

```
#include <cstdlib>
#include <cstdio>
#include <memory>
#include "../include/fsize.h"
#include "../include/error_count.h"
#include "../include/location.h"
#include "../include/trie.h"
#include "../include/scaner.h"
#include "../include/char_conv.h"
#include "../include/test_scaner.h"
#include "../include/errors_and_tries.h"
#include "../include/file_contents.h"
#include "../include/char_trie.h"

/* Функция, открывающая файл с тестовым текстом. Возвращает строку с текстом, если открыть
   файл удалось и размер файла не равен нулю, и пустую строку в противном случае. */
std::u32string init_testing(const char* name){
    auto contents = get_contents(name);
    auto str      = contents.second;
    switch(contents.first){
        case Get_contents_return_code::Normal:
            if(!str.length()){
                puts("Длина файла равна нулю.");
                return U"";
            }else{
                return utf8_to_u32string(str.c_str());
            }
            break;

        case Get_contents_return_code::Impossible_open:
            puts("Невозможно открыть файл.");
            return U"";

        case Get_contents_return_code::Read_error:
            puts("Ошибка при чтении файла.");
            return U"";
    }
    return U"";
}

int main(int argc, char** argv) {
    if(1 == argc){
        puts("Не задан тестовый файл.");
    }else{
        std::u32string t = init_testing(argv[1]);
        if(t.length()){
            char32_t* p   = const_cast<char32_t*>(t.c_str());
            auto      loc = std::make_shared<Location>(p);
```

```
            Errors_and_tries etr;
            etr.ec        = std::make_shared<Error_count>();
            etr.ids_trie  = std::make_shared<Char_trie>();
            etr.strs_trie = std::make_shared<Char_trie>();
            auto sc       = std::make_shared<Scaner>(loc,  etr);
            test_scaner(sc);
        }
    }
    return 0;
}
```

Пример файлов `test_scaner.cpp` и `test_scaner.h`:

```
/* Пример файла test_scaner.cpp. */
#include <cstdio>
#include "../include/test_scaner.h"
#include "../include/char_conv.h"

/* Данный массив состоит из строковых литералов, представляющих собой заключённые
 * в кавычки идентификаторы из перечисления Lexem_code. Строки идут в том же
 * порядке, что и соответствующие идентификаторы перечисления Lexem_code. */

static const char* lexem_names[] = {
    "Nothing",             "UnknownLexem",       "Action",
    "Opened_round_brack",  "Closed_round_brack", "Or",
    "Kleene_closure",      "Positive_closure",   "Optional_member",
    "Character",           "Begin_expression",   "End_expression",
    "Class_Latin",         "Class_Letter",       "Class_Russian",
    "Class_bdigits",       "Class_digits",       "Class_latin",
    "Class_letter",        "Class_odigits",      "Class_russian",
    "Class_xdigits",       "Class_ndq",          "Class_nsq",
    "M_Class_Latin",       "M_Class_Letter",     "M_Class_Russian",
    "M_Class_bdigits",     "M_Class_digits",     "M_Class_latin",
    "M_Class_letter",      "M_Class_odigits",    "M_Class_russian",
    "M_Class_xdigits",     "M_Class_ndq",        "M_Class_nsq"
};

void print_lexem(Expr_lexem_info li){
    Lexem_code lc = li.code;
    printf("%s", lexem_names[lc]);
    if(Character == lc){
        char32_t ch = li.c;
        if(ch < ' '){
            printf(" %u", ch);
        }else{
            std::string s = char32_to_utf8(ch);
            printf(" '%s'",s.c_str());
        }
    }
    printf(" \n");
}

void test_scaner(std::shared_ptr<Expr_scaner> sc){
    Lexem_info lexem;
    do{
        lexem = sc -> current_lexem();
        print_lexem(lexem);
```

```
    }while(lexem.code);
}

/* Пример файла test_scaner.h. */
#ifndef TEST_SCANER_H
#define        TEST_SCANER_H
#include <memory>
#include "../include/scaner.h"
void print_lexem(Lexem_info li);
void test_scaner(std::shared_ptr<Scaner> sc);
#endif        /* TEST_SCANER_H */
```

Список литературы

[1] Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий. 2–е изд.: Пер. с англ. — М.: Вильямс, 2008.

[2] Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. 2–е изд.: Пер. с англ. — М.: Вильямс, 2002.

[3] Фридл Дж. Регулярные выражения. 3–е изд: Пер. с англ. — СПб.: Символ-Плюс, 2008.

[4] Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. — СПб.: Наука и техника, 2006.

[5] Свердлов С.З. Языки программирования и методы трансляции. — СПб.: Питер, 2007.

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

[7] Креншоу Дж. Давайте создадим компилятор! — М.: 1995.

[8] Кнут Д.Э. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск.: Пер. с англ. — М.: Мир, 1978.

[9] Кнут Д.Э. Искусство программирования. Т.3. Сортировка и поиск. 2–е изд.: Пер. с англ. — М.: Вильямс, 2008.

[10] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы.: Пер с англ. — М.: Вильямс, 2000.