Регулярные выражения

Введение

Механизм *регулярных выражений* нашел применение во многих программных продуктах, входит в стандартную библиотеку или даже является частью языка программирования. При помощи регулярных выражений можно достаточно просто описать любой регулярный язык, а для регулярных языков существуют простые алгоритмы проверки принадлежности цепочки заданному языку.

Обзор задания: реализовать программу, которая принимает в качестве параметра регулярное выражение, читает строчки (цепочки символов) со стандартного ввода, а на стандартный вывод выдает только те строчки, которые удовлетворяют заданному регулярному выражению. У программы должен быть специальный режим работы, в котором она печатает на стандартный вывод описание сгенерированного для разбора цепочки автомата.

Теоретические сведения

Регулярные множества и выражения

Пусть T - конечный алфавит. *Регулярное множество* (*регулярный язык*) в алфавите T определяется рекурсивно следующим образом:

(1) {} (пустое множество) - регулярное множество в алфавите T;

(2) {a} - регулярное множество в алфавите T для каждого $a \in T$;

(3) {*e*} - регулярное множество в алфавите T (*e* - пустая цепочка);

(4) если P и Q - регулярные множества в алфавите T, то таковы же и множества

(а) $P \cup Q$ (объединение),

(б) PQ (конкатенация, т.е. множество $\{pq \mid p \in P, q \in Q\}$),

(в) P* (итерация: $P^* = \{e\} \cup P \cup PP \cup ...$);

(5) ничто другое не является регулярным множеством в алфавите T.

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо {}, либо {*e*}, либо {a} для некоторого $a \in T$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

Приведенное выше определение регулярного множества позволяет ввести следующую удобную форму его записи, называемую регулярным выражением.

Регулярное выражение в алфавите T и обозначаемое им регулярное множество в алфавите T определяются рекурсивно следующим образом:

(1)$\varnothing$ - регулярное выражение, обозначающее множество $\varnothing$;
(2)e - регулярное выражение, обозначающее множество {e};
(3)a - регулярное выражение, обозначающее множество {a};
(4)если p и q - регулярные выражения, обозначающие регулярные множества P и Q соответственно, то

1. (p|q) - регулярное выражение, обозначающее регулярное множество PQ,
2. (pq) - регулярное выражение, обозначающее регулярное множество $P \cup Q$,
3. (p^*) - регулярное выражение, обозначающее регулярное множество P^*;

(5)ничто другое не является регулярным выражением в алфавите T.

Мы будем опускать лишние скобки в регулярных выражениях, договорившись о том, что операция итерации имеет наивысший приоритет, затем идет операции конкатенации, наконец, операция объединения имеет наименьший приоритет.

Кроме того, мы будем пользоваться записью p^+ для обозначения pp^*. Таким образом, запись (a|((ba)(a^*))) эквивалентна a|ba^+.

Наконец, мы будем использовать запись L(r) для регулярного множества, обозначаемого регулярным выражением r.

Пример. Несколько примеров регулярных выражений и обозначаемых ими регулярных множеств:

- a(e|a)|b - обозначает множество {a, b, aa};
- $a(a|b)^*$ - обозначает множество всевозможных цепочек, состоящих из a и b, начинающихся с a;
- $(a|b)^*(a|b)(a|b)^*$ - обозначает множество всех непустых цепочек, состоящих из a и b, т.е. множество $\{a, b\}^+$;
- $((0|1)(0|1)(0|1))^*$ - обозначает множество всех цепочек, состоящих из нулей и единиц, длины которых делятся на 3.

Ясно, что для каждого регулярного множества можно найти регулярное выражение, обозначающее это множество, и наоборот. Более того, для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений.

Будем говорить, что регулярные выражения равны или эквивалентны (=), если они обозначают одно и то же регулярное множество.

Существует ряд алгебраических законов, позволяющих осуществлять эквивалентное преобразование регулярных выражений.

Лемма. Пусть p, q и r - регулярные выражения. Тогда справедливы следующие соотношения:

(1) p|q = q|p; (7) pe = ep = p;
(2) $\varnothing^* = e$; (8) $\varnothing p = p\varnothing = \varnothing$;
(3) p|(q|r) = (p|q)|r; (9) $p^* = p|p^*$;
(4) p(qr) = (pq)r; (10) $(p^*)^* = p^*$;
(5) p(q|r) = pq|pr; (11) p|p = p;
(6) (p|q)r = pr|qr; (12) $p|\varnothing = p$.

Следствие. Для любого регулярного выражения существует эквивалентное регулярное выражение, которое либо есть ∅, либо не содержит в своей записи ∅.

В дальнейшем будем рассматривать только регулярные выражения, не содержащие в своей записи ∅.

Конечные автоматы

Регулярные выражения, введенные ранее, служат для описания регулярных множеств. Для распознавания регулярных множеств служат конечные автоматы.

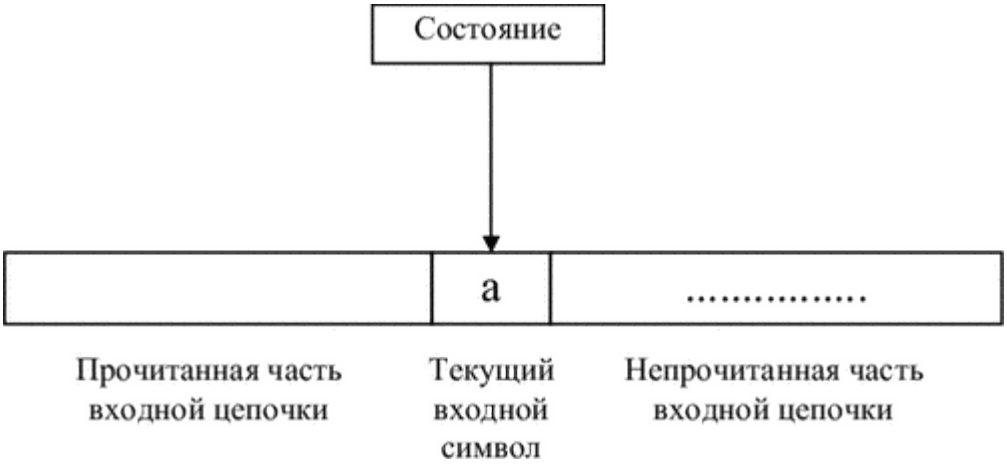
Недетерминированный конечный автомат (НКА) - это пятерка M = (Q, T, D, q_0, F), где

- Q - конечное множество состояний;
- T - конечное множество допустимых входных символов (входной алфавит);
- D - функция переходов (отображающая множество $Q\times(T\cup\{e\})$ во множество подмножеств

множества Q), определяющая поведение управляющего устройства;

- $q_0 \in Q$ - начальное состояние управляющего устройства;
- $F \subseteq Q$ - множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и, возможно, сдвига входной головки на одну ячейку вправо:



Недетерминизм автомата заключается в том, что, во-первых, находясь в некотором состоянии и обозревая текущий символ, автомат может перейти в одно из, вообще говоря, нескольких возможных состояний, и во-вторых, автомат может делать переходы по e.

Пусть M = (Q, T, D, q_0, F) - НКА. Конфигурацией автомата M называется пара (q, w) $\in Q \times T^*$, где q - текущее состояние управляющего устройства, а w - цепочка символов на входной ленте, состоящая из символа под головкой и всех символов справа от него. Конфигурация (q_0, w) называется начальной, а конфигурация (q, e), где $q \in F$ - заключительной (или допускающей).

Пусть M = (Q, T, D, q_0, F) - НКА. Тактом автомата M называется бинарное отношение $\vdash$, определенное на конфигурациях M следующим образом: если $p \in D(q, a)$, где $a \in T \cup \{e\}$, то (q, aw) $\vdash$ (p, w) для всех $w \in T^*$.

Будем обозначать символом $\vdash^+$ ($\vdash^*$) транзитивное (рефлексивно-транзитивное) замыкание отношения $\vdash$.

Говорят, что автомат M допускает цепочку w, если (q_0, w) $\vdash^*$(q, e) для некоторого $q \in F$. Языком, допускаемым (распознаваемым, определяемым) автоматом M, (обозначается L(M)), называется множество входных цепочек, допускаемых автоматом M. Т.е.

$$L(M) = \{w | w \in T^* \text{ и } (q_0, w) \vdash^* (q, e) \text{ для некоторого } q \in F\}.$$

Важным частным случаем недетерминированного конечного автомата является детерминированный конечный автомат, который на каждом такте работы имеет возможность перейти не более чем в одно состояние и не может делать переходы по e.

Пусть M = (Q, T, D, q_0, F) - НКА. Будем называть M детерминированным конечным автоматом (ДКА), если выполнены следующие два условия:

- $D(q, e) = \varnothing$ для любого $q \in Q$, и
- D(q, a) содержит не более одного элемента для любых $q \in Q$ и $a \in T$.

Так как функция переходов ДКА содержит не более одного элемента для любой пары аргументов, для ДКА мы будем пользоваться записью D(q, a) = p вместо D(q, a) = {p}.

Конечный автомат может быть изображен графически в виде диаграммы, представляющей собой ориентированный граф, в котором каждому состоянию соответствует вершина, а дуга, помеченная символом $a \in T \cup \{e\}$, соединяет две вершины p и q, если $p \in D(q, a)$. На диаграмме выделяются начальное и заключительные состояния (в примерах ниже, соответственно, входящей стрелкой и двойным контуром).

Пример. Пусть L = L(r), где $r = (a|b)^*a(a|b)(a|b)$.

- Недетерминированный конечный автомат M, допускающий язык L:

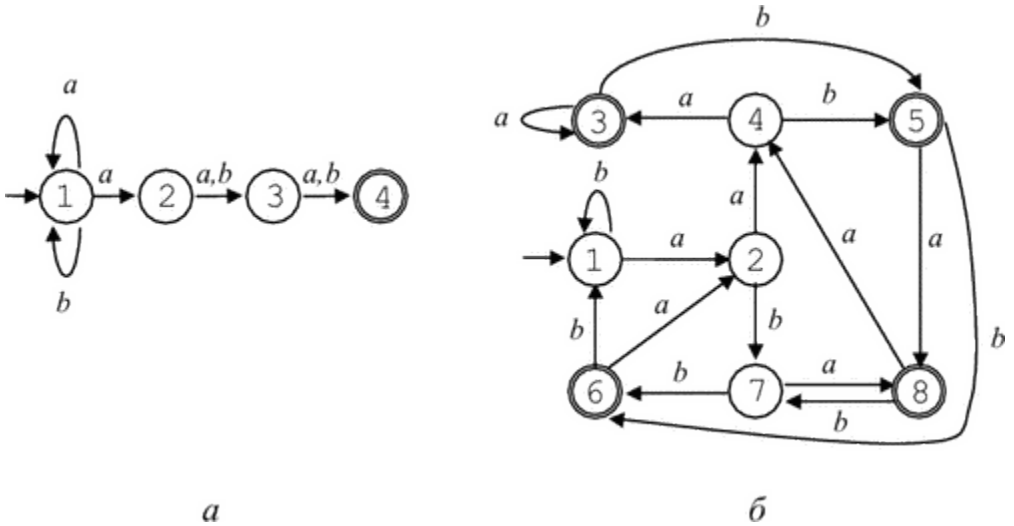
  M = {{1, 2, 3, 4}, {a, b}, D, 1, {4}},
  где функция переходов D определяется так:

  D(1, a) = {1, 2}, D(3, a) = {4},
  D(2, a) = {3}, D(3, b) = {4},
  D(2, b) = {3}.
  Диаграмма автомата приведена на рисунке слева.

- Детерминированный конечный автомат M, допускающий язык L:

  M = {{1, 2, 3, 4, 5, 6, 7, 8}, {a, b}, D, 1, {3, 5, 6, 8}},
  где функция переходов D определяется так:

  D(1, a) = 2, D(5, a) = 8,
  D(1, b) = 1, D(5, b) = 6,
  D(2, a) = 4, D(6, a) = 2,
  D(2, b) = 7, D(6, b) = 1,
  D(3, a) = 3, D(7, a) = 8,
  D(3, b) = 5, D(7, b) = 6,
  D(4, a) = 3, D(8, a) = 4,
  D(4, b) = 5, D(8, b) = 7.
  Диаграмма автомата приведена на рисунке справа.



а *б*

Пример. Анализ цепочек.

- При анализе цепочки w = ababa автомат **a** из примера может сделать следующую

последовательность тактов:

(1, ababa) ⊢ (1, baba) ⊢ (1, aba) ⊢ (2, ba) ⊢ (3, a) ⊢ (4, e).

Состояние 4 является заключительным, следовательно, цепочка w допускается этим автоматом.

- При анализе цепочки w = ababab автомат **б** из примера должен сделать следующую последовательность тактов:

  (1, ababab) ⊢ (2, babab) ⊢ (7, abab) ⊢ (8, bab) ⊢ (7, ab) ⊢ (8, b) ⊢ (7, e).

  Так как состояние 7 не является заключительным, цепочка w не допускается этим автоматом.

Построение НКА по регулярному выражению

Рассмотрим алгоритм построения по регулярному выражению недетерминированного конечного автомата, допускающего тот же язык.

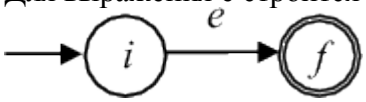
Алгоритм. Построение недетерминированного конечного автомата по регулярному выражению.

Вход. Регулярное выражение r в алфавите T.

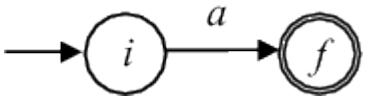
Выход. НКА M, такой что L(M) = L(r).

Метод. Автомат для выражения строится композицией из автоматов, соответствующих подвыражениям. На каждом шаге построения строящийся автомат имеет в точности одно заключительное состояние, в начальное состояние нет переходов из других состояний и нет переходов из заключительного состояния в другие.

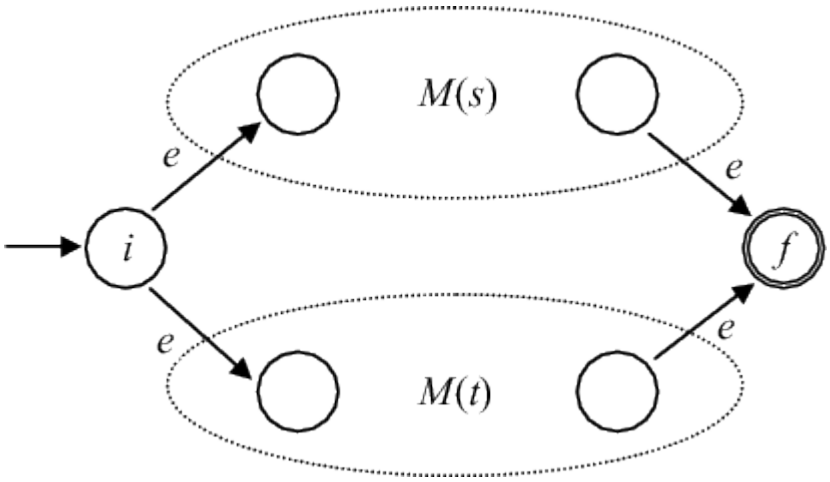
- Для выражения e строится автомат



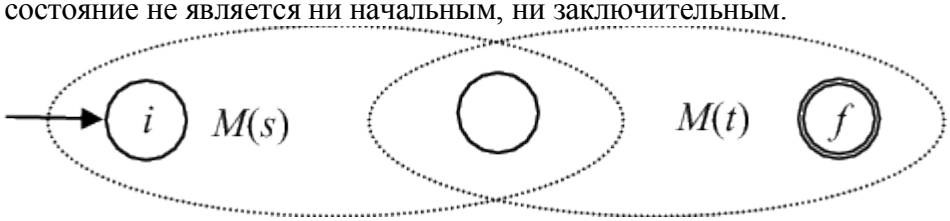
- ∈



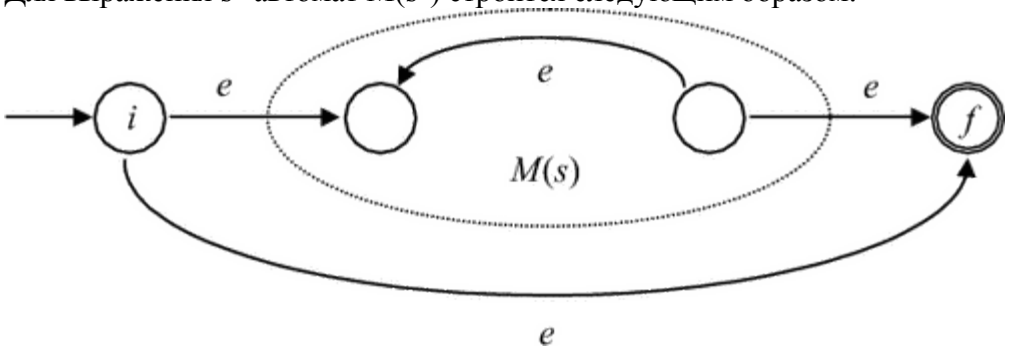
- Для выражения s|t автомат M(s|t) строится как показано на рисунке. Здесь i - новое начальное состояние и f - новое заключительное состояние. Заметим, что имеет место переход по e из i в начальные состояния M(s) и M(t) и переход по e из заключительных состояний M(s) и M(t) в f. Начальное и заключительное состояния автоматов M(s) и M(t) не являются таковыми для автомата M(s|t).



- Для выражения st автомат M(st) строится следующим образом: начальное состояние M(s) становится начальным для нового автомата, а заключительное состояние M(t) становится заключительным для нового автомата. Начальное состояние M(t) и заключительное состояние M(s) сливаются, т.е. все переходы из начального состояния M(t) становятся переходами из заключительного состояния M(s). В новом автомате это объединенное состояние не является ни начальным, ни заключительным.



- Для выражения s^* автомат $M(s^*)$ строится следующим образом:



Здесь i - новое начальное состояние, а f - новое заключительное состояние.

Построение ДКА по НКА

Рассмотрим алгоритм построения по недетерминированному конечному автомату детерминированного конечного автомата, допускающего тот же язык.

Алгоритм. Построение детерминированного конечного автомата по недетерминированному.

Вход. НКА M = (Q, T, D, q_0, F).

Выход. ДКА M' = (Q', T, D', q_0', F'), такой что L(M) = L(M').

Метод. Каждое состояние результирующего ДКА - это некоторое множество состояний исходного НКА.

В алгоритме будут использоваться следующие функции:

e-closure(R) ($R \subseteq Q$) - множество состояний НКА, достижимых из состояний, входящих в R, посредством только переходов по e, т.е. множество

$$S = \bigcup_{q \in R} \{p | (q, e) \vdash^* (p, e)\}$$

move(R, a) ($R \subseteq Q$) - множество состояний НКА, в которые есть переход на входе a для состояний из R, т.е. множество

$$S = \bigcup_{q \in R} \{p | p \in D(q, a)\}$$

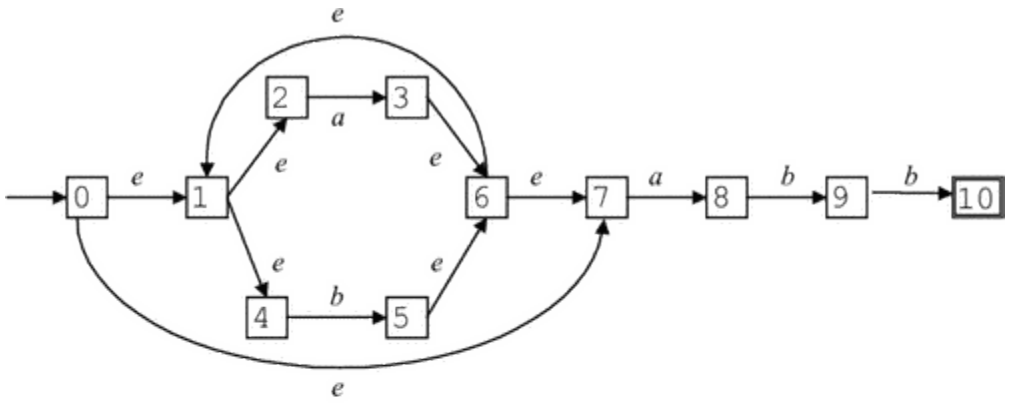
Вначале Q' и D' пусты. Выполнить шаги 1-4:

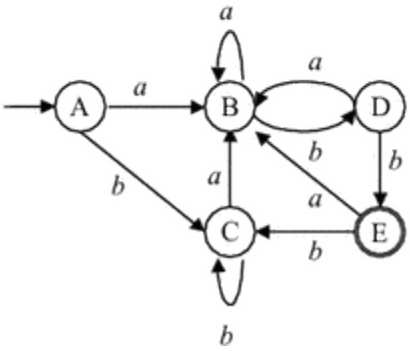
- Определить q_0' = e-closure($\{q_0\}$).
- Добавить q_0' в Q' как непомеченное состояние.
- Выполнить следующую процедуру:

```
while (в Q' есть непомеченное состояние R)
{
        пометить R;
        for (каждого входного символа a ∈ T)
        {
                S = e-closure(move(R, a));
                        if (S≠∅)
                        {
                                if (S∉Q')
                                        добавить S в Q' как непомеченное состояние;
                                определить D'(R, a) = S;
                        }
        }
}
```

- Определить F' = {S|S ∈ Q', S ∩ F≠∅}.

Пример. Применение данного алгоритма:





Построение ДКА по регулярному выражению

Приведем теперь алгоритм построения по регулярному выражению детерминированного конечного автомата, допускающего тот же язык.

Пусть дано регулярное выражение r в алфавите T. К регулярному выражению r добавим маркер конца: (r)#. Такое регулярное выражение будем называть пополненным. В процессе своей работы алгоритм будет использовать пополненное регулярное выражение.

Алгоритм будет оперировать с синтаксическим деревом для пополненного регулярного выражения (r)# , каждый лист которого помечен символом a $\in$ T $\cup$ {e, #}, а каждая внутренняя вершина помечена знаком одной из операций: **.** (конкатенация), | (объединение), * (итерация).

Каждому листу дерева (кроме e-листьев) припишем уникальный номер, называемый позицией, и будем использовать его, с одной стороны, для ссылки на лист в дереве, и, с другой стороны, для ссылки на символ, соответствующий этому листу. Заметим, что если некоторый символ используется в регулярном выражении несколько раз, он имеет несколько позиций.

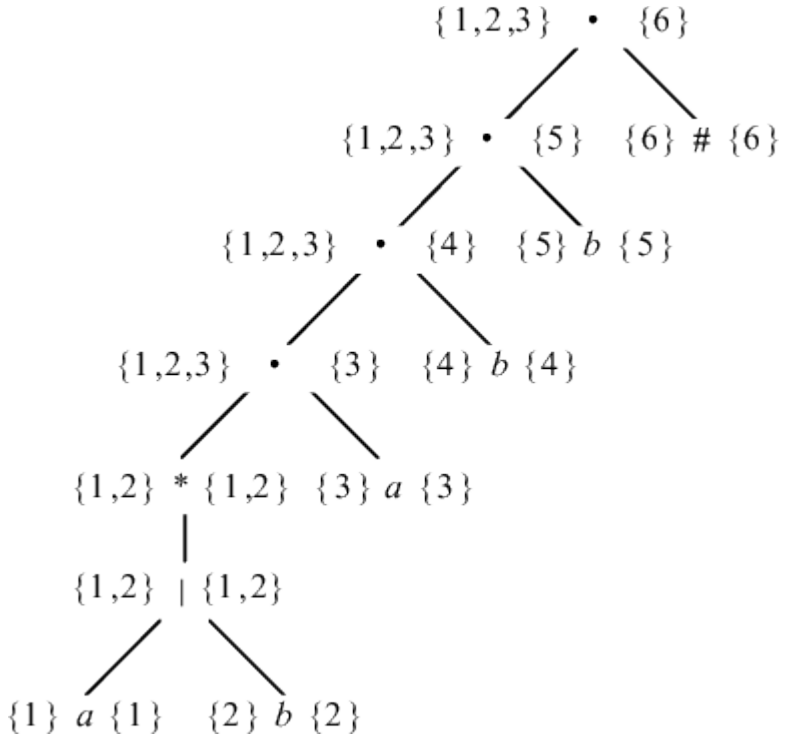
Теперь, обходя дерево T снизу-вверх слева-направо, вычислим четыре функции: nullable, firstpos, lastpos и followpos. Функции nullable, firstpos и lastpos определены на узлах дерева, а followpos - на множестве позиций. Значением всех функций, кроме nullable, является множество позиций. Функция followpos вычисляется через три остальные функции.

Функция firstpos(n) для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в n. Аналогично, lastpos(n) дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной n. Для узла n, поддеревья которого (т.е. деревья, у которых узел n является корнем) могут породить пустое слово, определим nullable(n) = true, а для остальных узлов nullable(n) = false.

Таблица для вычисления функций nullable, firstpos и lastpos:

узел n	nullable(n)	furstpos(n)	lastpos(n)
лист e	true	∅	∅
лист i	false	{i}	{i}
| ∧ u v	nullable(u) or nullable(v)	firstpos(u) ∪ firstpos(v)	lastpos(u) ∪ lastpos(v)
. ∧ u v	nullable(u) and nullable(v)	if nullable(u) firstpos(u) ∪ firstpos(v) else firstpos(u)	if nullable(v) lastpos(u) ∪ lastpos(v) else lastpos(v)
* | u	true	firstpos(u)	lastpos(u)

Пример. Синтаксическое дерево для пополненного регулярного выражения $(a|b)^*abb\#$ с результатом вычисления функций firstpos и lastpos приведено на рисунке ниже. Слева от каждого узла расположено значение firstpos, справа от узла - значение lastpos. Заметим, что эти функции могут быть вычислены за один обход дерева.



Если i - позиция, то followpos(i) есть множество позиций j таких, что существует некоторая строка ...cd..., входящая в язык, описываемый регулярным выражением, такая, что позиция i соответствует этому вхождению c, а позиция j - вхождению d.

Функция followpos может быть вычислена также за один обход дерева снизу-вверх по следующим двум правилам.

1. Пусть n - внутренний узел с операцией **.** (конкатенация), u и v - его потомки. Тогда для каждой позиции i, входящей в lastpos(u), добавляем к множеству значений followpos(i) множество firstpos(v).

2. Пусть n - внутренний узел с операцией * (итерация), u - его потомок. Тогда для каждой позиции

i, входящей в lastpos(u), добавляем к множеству значений followpos(i) множество firstpos(u).

Пример. Результат вычисления функции followpos для регулярного выражения из предыдущего примера:

позиция	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	∅

Алгоритм. Прямое построение ДКА по регулярному выражению.

Вход. Регулярное выражение r в алфавите T.

Выход. ДКА M = (Q, T, D, q_0, F), такой что L(M) = L(r).

Метод. Состояния ДКА соответствуют множествам позиций.

Вначале Q и D пусты. Выполнить шаги 1-6:

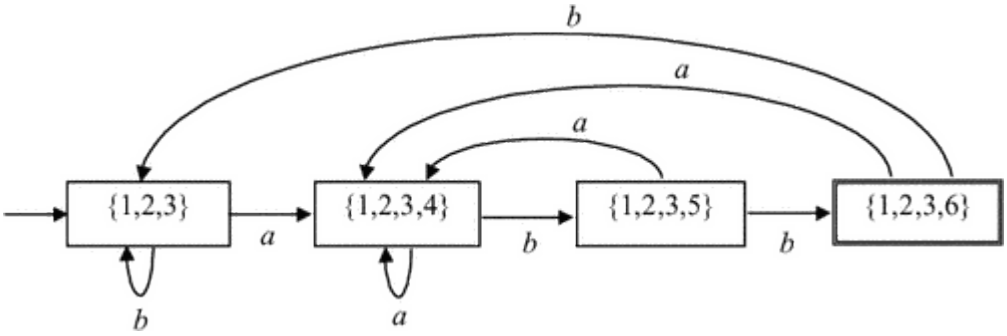
- Построить синтаксическое дерево для пополненного регулярного выражения (r)#.
- Обходя синтаксическое дерево, вычислить значения функций nullable, firstpos, lastpos и followpos.
- Определить q_0 = firstpos(root), где root - корень синтаксического дерева.
- Добавить q_0 в Q как непомеченное состояние.
- Выполнить следующую процедуру:

```
while (в Q есть непомеченное состояние R)
{
        пометить R;
        for (каждого входного символа a ∈ T , такого, что в R имеется позиция, которой
        соответствует a)
        {
                пусть символ a в R соответствует позициям p1, ..., pn,
                и пусть S = ∪ 1≤i≤n followpos(pi);
                if (S≠∅)
                {
                        if (S⊄Q)
                                добавить S в Q как непомеченное состояние;
                        определить D(R, a) = S;
                }
        }
}
```

- Определить F как множество всех состояний из Q, содержащих позиции, связанные с

символом #.

Пример. Результат применения алгоритма для регулярного выражения $(a|b)^*abb$ приведен на рисунке ниже:



Задание

Базовое задание

Реализовать программу на языке C++, которая принимает на вход регулярное выражение, проверяет его корректность, строит по регулярному выражению детерминированный конечный автомат, после чего фильтрует стандартный поток ввода, копируя в стандартный поток вывода только те строчки, которые принадлежат регулярному языку, задаваемому регулярным выражением.

Формат командной строки

```
./regexp [-a] регулярное_выражение
```

Если флаг -a не был передан программе, то она работает в стандартном режиме: в стандартный поток вывода копируются только те строчки из стандартного потока ввода, которые удовлетворяют регулярному выражению, переданному в качестве параметра. Программа завершает работу при обнаружении сигнала конца файла.

Если задан флаг -a, программа должна распечатать в поток стандартного вывода сгенерированный для данного регулярного выражения детерминированный конечный автомат и завершить работу.

В любом случае, если переданное в качестве параметра регулярное_выражение не является корректным регулярным выражением, программа должна распечатать сообщение об ошибке и завершить работу досрочно.

Замечание. Многие символы, используемые при записи регулярного выражение также являются метасимволами shell, поэтому параметр программы при запуске лучше всего заключать в одинарные кавычки, пример:

```
./regexp '(ab|a*)cd'
```

Формат вывода сгенерированного конечного автомата

Вывод начинается со строки “States:”, за которой следует список состояний. Начальное состояние помечается добавлением “(S)” к имени состояния, а конечные состояния помечаются символами “(F)”. Далее следует строка “Transitions:”, за которой следует описание переходов. Каждый переход описывается следующим образом: “состояние, символ -> новое состояние”.

Пример. Вывод для автомата из раздела “Построение ДКА по регулярному выражению должен выглядеть следующим образом:

```
States:
{1,2,3} (S)
{1,2,3,4}
{1,2,3,5}
{1,2,3,6} (F)
Transitions:
{1,2,3},b -> {1,2,3}
{1,2,3},a -> {1,2,3,4}
{1,2,3,4},a -> {1,2,3,4}
{1,2,3,4},b -> {1,2,3,5}
{1,2,3,5},a -> {1,2,3,4}
{1,2,3,5},b -> {1,2,3,6}
{1,2,3,6},a -> {1,2,3,4}
{1,2,3,6},b -> {1,2,3}
```

Формат записи регулярного выражения

Общий формат регулярных выражений описан в разделе “Регулярные множества и выражения”. Символы '(', ')', '|', '*' являются специальными в записи регулярного выражения. Все остальные символы должны рассматриваться как часть алфавита языка, задаваемого регулярным выражением. Если возникает необходимость включить символы '(', ')', '|', '*' в качестве символов алфавита языка, они должны быть записаны в регулярном выражении при помощи escape-последовательности с символом '\', то есть как '\(', '\)', '\|', '*'. Сам символ '\' должен быть записан как '\\'.

Примеры корректных регулярных выражений: \((a|b)*\), (aba)\+(a*|b*), ((0|1|2|3)*)*, <пустая строка>, *, ***, (|a|*)*.

Примеры некорректных регулярных выражений: def), (abc.

Замечание. Если по дополнительных вариантов грамматика должна быть расширена, то количество специалных символов может увеличиться, так специальными могут стать следующие символы: '+', '[', ']', '{', '}', '.'. При этом символы '-' и ',' могут считаться специальными только внутри квадратных и фигурных скобок, соответственно.

Замечание. Можно считать что нулевой символ ('\0') не встречается никогда ни в регулярном выражении, ни в стандартном потоке ввода программы. Поэтому нулевой символ может использоваться, например, при описании в памяти недетерминированного конечного автомата для представления перехода по символу *e* (пустая цепочка).

Пример работы программы

(Здесь вывод программы отмечен **жирным** шрифтом.)

```
./regexp '(a|b|c)d*(e|f|g)'
addde
addde
bb
be
be
cde
cde
ddddf
```

Рекомендуемые этапы реализации

1. Составить описание контекстно-свободной (УКС) грамматики, описывающий язык регулярных выражений. Грамматика должна учитывать приоритет операций. Полученная грамматика может оказаться непригодной для применения метода рекурсивного спуска в чистом виде, более того, она может неоднозначной. Используя рекомендации из “Формальные грамматики и языки. Элементы теории трансляции” по преобразованию грамматик для применения метода рекурсивного спуска, а также здравый смысл, придумать, как метод рекурсивного спуска может быть применен (в грамматике могут появиться правила, для которых выбор альтернативы может быть неоднозначным в чистом виде, но на основе вида следующей лексемы можно осуществить выбор нужной альтернативы).
2. Реализовать лексический анализатор для языка регулярных выражений. Лексический анализатор должен отделять специальные символы регулярных выражений от всех остальных, обрабатывать escape-последовательности (например, '\(').
3. Методом рекурсивного спуска реализовать анализатор грамматики регулярных выражений. В зависимости от варианта задания, анализатор может во время разбора создавать дерево регулярного выражения или сразу строить НКА (для варианта I).
4. Реализовать, в зависимости от варианта, необходимые для получения ДКА преобразования.
5. Реализовать метод, “запускающий” ДКА на входной строке и выдающий результат: допускает ли ДКА цепочку или нет (принадлежит ли цепочка языку, задаваемому регулярным выражением).

Варианты

Основные варианты

I. Построение ДКА по регулярному выражению осуществляется путём получения НКА и дальнейшим преобразованием НКА в ДКА.

II. Прямое построение ДКА по регулярному выражению (с использованием функции followpos).

Дополнительные варианты

a. Расширение грамматики регулярных выражений символом +: (x+) считается эквивалентным (xx*). Приоритет операции + считается равным приоритету операции *.

b. Расширение грамматики регулярных выражений конструкцией $[a_1\text{-}a_2]$, которая считается эквивалентной конструкции $(b_1|b_2|...|b_n)$, где $b_1=a_1$, $b_n=a_2$, коды символов b_i и b_{i+1} отличаются ровно на единицу. Пример: [0-9] = (0|1|2|3|4|5|6|7|8|9). Ситуация, когда код символа a_1 больше кода символа a_2, считается ошибочной. Приоритет конструкции $[a_1\text{-}a_2]$ считается равным приоритету символа.

c. Расширение грамматики регулярных выражений конструкцией {i,j}, которая означает минимальное и максимальное количество повторений конструкции, следующей перед ней. Конструкция x{i,j} считается эквивалентной конструкции $(x^i|x^{i+1}|...|x^j)$, где x^k обозначает k-кратное повторение конструкции x. Пример: (a|b){0,3}, что означает повторение от 0 (пустая цепочка) до 3 раз символа a или символа b. Ситуация, когда i или j меньше нуля или когда i > j, считается ошибочной. Приоритет конструкции {i,j} считается равным приоритету операции *.

d. Реализовать конструкцию '.', которая в записи регулярного выражения обозначает *любой символ*. Пример: .*, что означает “любая цепочка символов”.

Замечания по реализации на языке C++

Результатом реализации задачи должна быть не только работающая программа, но и продуманный набор классов для решения задачи. В программе по возможности должны быть использованы только возможности стандартной библиотеки C++, без стандартной библиотеки C (cin, cout, cerr вместо fgets(), printf(), fprintf() и т.п.). Приветствуется широкое использование возможностей STL: например, контейнерных шаблонов vector, deque, map, set, алгоритмов, класса string.

Возможная иерархия классов для решения данной задачи:

1. FA – конечный автомат и два его потомка NFA (НКА) и DFA (ДКА). (NFA только в I варианте).
2. LexAnalyzer – лексический анализатор.
3. SyntaxAnalyzer – грамматический анализатор, реализованный методом рекурсивного спуска.
4. RegexpNode – абстрактный предок всех узлов дерева разбора регулярного выражения (обязательно во II варианте, возможно и в I варианте). UnRegexpNode – потомок RegexpNode, узел с одной дочерней вершиной. BiRegexpNode – потомок RegexpNode, узел с двумя дочерними вершинами. AsteriskNode – потомок UnRegexpNode, узел '*'. ConcatNode – потомок BiRegexpNode, узел конкатенации ('.'). PipeNode – потомок Biregexp Node, узел '|'. CharNode – потомок RegexpNode, лист дерева с символом. EmptyNode – потомок RegexpNode, узел дерева с символом *e*. В зависимости от варианта задания могут быть добавлены узлы для других операций.
5. Tree2DFA – класс, осуществляющий преобразование дерева регулярного выражения в ДКА напрямую (II вариант).
6. В I варианте НКА может строиться во время разбора выражения или по построенному дереву классом Tree2NFA.
7. NFA2DFA – класс преобразования НКА в ДКА (I вариант).