Формальные языки и трансляции
Практикум №1
Демешко Алеся, 795 группа
Регулярные выражения

В каждой задаче нужно реализовать на языке C++ или Python. Некоторый алгоритм обработки регулярных выражений. В каждой задаче аргументами являются строка в алфавите $\{a, b, c, 1, \cdot, +, *\}$, а также некоторые дополнительные параметры. Если задача предполагает ответ "да/нет то необходимо вывести yes в случае положительного ответа и no - в случае отрицательного. В случае, если ответ является целым числом или словом, необходимо вывести это число или слово. В случае, если таких числа или слова не существует, необходимо вывести INF. В случае, если входная строка не является корректным регулярным выражением в обратной польской записи, необходимо выдать сообщение error об ошибке. Дополнительные случаи оговорены непосредственно при формулировке задачи. В дальнейшем предполагается, что первым компонентом входа является регулярное выражение $\alpha$ в обратной польской записи, задающее язык $L$. Вариант 19: Даны $a$, буква $x$. Найти максимальное $k$, такое что в $L$ есть слова, заканчивающиеся на $x^k$.

Решение: Для решения задачи нам понадобятся два стека: stackSuf - стек разбора выражения в котором для каждой части хранится длина максимального суффикса из $x$ и stackLenOfWhole – стек разбора выражения в котором для каждой части хранится длина максимальной строки состоящей только из x. Далее идём по выражению.

1) Если мы получаем на вход буквы $\{a, b, c\}$, мы сравниваем ее с $x$ и, если равна, то длина и максимального слова, и суффикса равны 1, соответственно добавляем в стеки по 1. Иначе – длина максимального слова будет равна -1, а суффикс 0.

2) Если мы получаем на вход $\cdot$, то максимальная длина слова равна сумме двух максимальных длин членов операции, а максимальный суффикс равен или старому суффиксу второго члена, или длине всего второго члена плюс максимальный суффикс первого члена. То есть мы рассматриваем два случая и выбираем из них максимальный.

3) Если мы получаем на вход $*$, то длина максимального суффикса останется без изменения, а максимального слова станет равна бесконечности, если до этого она была положительна, иначе – поместим в стек -1.

4) Если мы получаем на вход $+$, то каждое из значений - максимум из слагаемых.

Код на Python для решения задачи

```
1 regular = input()
2 regular = list(regular)
3 x = input()
4 print(regular)
5 stackSuf = []   # стек разбора выражения в котором для каждой части хранится длина
      максимального суффикса из x
```

```
6  INF = 1000000000000000000
7  stackLenOfWhole = []   # стек разбора выражения в котором для каждой части хранится длина
       максимальной строки
8  # состоящей только из x
9  for symb in regular:
10     if symb == '.':
11         SufFirst = stackSuf.pop()
12         SufSecond = stackSuf.pop()
13         LenOfWhxoleFirst = stackLenOfWhole.pop()
14         LenOfWholeSecond = stackLenOfWhole.pop()
15         stackSuf.append(max(SufFirst, LenOfWholeFirst + SufSecond))   # рассматриваем
               два случая, оставить суффикс тот же
16         # или сделать вторую строку только из x, а в первой
17         # взять максимальный суффикс
18         stackLenOfWhole.append(LenOfWholeFirst + LenOfWholeSecond)
19     elif symb == '*':
20         SufFirst = stackSuf.pop()
21         LenOfWholeFirst = stackLenOfWhole.pop()
22         if LenOfWholeFirst > 0:   # если длина максимальной строки из x ненулевая, то оба
               параметра становятся бесконечностью
23             stackSuf.append(INF)
24             stackLenOfWhole.append(INF)
25         else:   # иначе параметры не меняются
26             stackSuf.append(SufFirst)
27             stackLenOfWhole.append(-1)
28     elif symb == '+':
29         SufFirst = stackSuf.pop()
30         SufSecond = stackSuf.pop()
31         LenOfWholeFirst = stackLenOfWhole.pop()
32         LenOfWholeSecond = stackLenOfWhole.pop()
33         stackSuf.append(max(SufFirst, SufSecond))   # оба параметра меняются на
               максимальный в случае плюса
34         stackLenOfWhole.append(max(LenOfWholeFirst, LenOfWholeSecond))
35     else:   # иначе просто добавляем букву
36         if symb == x:
37             stackSuf.append(1)
38             stackLenOfWhole.append(1)
39         else:
40             stackSuf.append(0)
41             stackLenOfWhole.append(-1)
42 last = stackSuf.pop()
43 if last == INF:
44     print("INF")
45 else:
46     print(last)
```

Приведенный код правильно работает на тестах, предоставленных лектором.

In [1]:
```
regular = input()
regular = list(regular)
x = input()
stackSuf = []  # стек разбора выражения в котором для каждой части хранится длина максимального суффикса из x
INF = 1000000000000000000
stackLenOfWhole = []  # стек разбора выражения в котором для каждой части хранится длина максимальной строки
# состоящей только из x
for symb in regular:
    if symb == '.':
        SufFirst = stackSuf.pop()
        SufSecond = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        LenOfWholeSecond = stackLenOfWhole.pop()
        stackSuf.append(max(SufFirst, LenOfWholeFirst + SufSecond))  # рассматриваем два случая, оставить суффи
        # или сделать вторую строку только из x, а в первой
        # взять максимальный суффикс
        stackLenOfWhole.append(LenOfWholeFirst + LenOfWholeSecond)
    elif symb == '*':
        SufFirst = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        if LenOfWholeFirst > 0:  # если длина максимальной строки из x ненулевая, то оба параметра становятся бе
            stackSuf.append(INF)
            stackLenOfWhole.append(INF)
        else:  # иначе параметры не меняются
            stackSuf.append(SufFirst)
            stackLenOfWhole.append(-1)
    elif symb == '+':
        SufFirst = stackSuf.pop()
        SufSecond = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        LenOfWholeSecond = stackLenOfWhole.pop()
        stackSuf.append(max(SufFirst, SufSecond))  # оба параметра меняются на максимальный в случае плюса
        stackLenOfWhole.append(max(LenOfWholeFirst, LenOfWholeSecond))
    else:  # иначе просто добавляем букву
        if symb == x:
            stackSuf.append(1)
            stackLenOfWhole.append(1)
        else:
            stackSuf.append(0)
            stackLenOfWhole.append(-1)
last = stackSuf.pop()
if last == INF:
    print("INF")
else:
    print(last)
```

```
ab+c.aba.*.bac.+.+*
b
1
```

In [1]:
```
regular = input()
regular = list(regular)
x = input()
stackSuf = []  # стек разбора выражения в котором для каждой части хранится длина максимального суффикса из x
INF = 1000000000000000000
stackLenOfWhole = []  # стек разбора выражения в котором для каждой части хранится длина максимальной строки
# состоящей только из x
for symb in regular:
    if symb == '.':
        SufFirst = stackSuf.pop()
        SufSecond = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        LenOfWholeSecond = stackLenOfWhole.pop()
        stackSuf.append(max(SufFirst, LenOfWholeFirst + SufSecond))  # рассматриваем два случая, оставить суф
        # или сделать вторую строку только из x, а в первой
        # взять максимальный суффикс
        stackLenOfWhole.append(LenOfWholeFirst + LenOfWholeSecond)
    elif symb == '*':
        SufFirst = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        if LenOfWholeFirst > 0:  # если длина максимальной строки из x ненулевая, то оба параметра становятся
            stackSuf.append(INF)
            stackLenOfWhole.append(INF)
        else:  # иначе параметры не меняются
            stackSuf.append(SufFirst)
            stackLenOfWhole.append(-1)
    elif symb == '+':
        SufFirst = stackSuf.pop()
        SufSecond = stackSuf.pop()
        LenOfWholeFirst = stackLenOfWhole.pop()
        LenOfWholeSecond = stackLenOfWhole.pop()
        stackSuf.append(max(SufFirst, SufSecond))  # оба параметра меняются на максимальный в случае плюса
        stackLenOfWhole.append(max(LenOfWholeFirst, LenOfWholeSecond))
    else:  # иначе просто добавляем букву
        if symb == x:
            stackSuf.append(1)
            stackLenOfWhole.append(1)
        else:
            stackSuf.append(0)
            stackLenOfWhole.append(-1)
last = stackSuf.pop()
if last == INF:
    print("INF")
else:
    print(last)
```

```
acb..bab.c.*.ab.ba.+.+*a.
b
0
```

Код на C++ для построения автомата по заданному регулярному выражению (выводит список смежности и список терминальных вершин

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  class Automata{
6  public:
7      explicit Automata(char symb) {
8          avt.push_back({});
9          avt[0].push_back({symb, 1});
10         avt.push_back({});
11         term.push_back(1);
12     }
13     vector<vector<pair<char, int>>> avt;   // автомат
14     vector<int> term; // терминальные вершины
15 };
16
17 void star(Automata &A) {
18     bool isRoot = false;   // встречался ли корень в терминальных вершинах
19     for(int i = 0;i < (int)A.term.size();++i) {
20         if(A.term[i] == 0) isRoot = true;
21         A.avt[A.term[i]].push_back({' ', 0});   // добавляем ребро от каждой
               терминальной вершины в корень
22     }
23     if(!isRoot) {
24         A.term.push_back(0);
```

```
25      }
26  }
27  Automata point(Automata &first , Automata &second) {
28      Automata result = first;
29      result.term.clear();
30      int size = 0;
31      for(int i = 0;i < (int)first.term.size();++i) {
32          size = (int)result.avt.size();
33          --size;
34          for(int j = 0;j < (int)second.avt[0].size();++j) {
35              result.avt[first.term[i]].push_back({second.avt[0][j].first , second.avt
                    [0][j].second + size}); // подвешиваем к каждой
                    вершине
36              // детей корня второго
37          }
38          for(int j = 1;j < (int)second.avt.size();++j) {  // добавляем все остальные
                вершины в автомат, при этом у каждой обновляем индекс
39              vector<pair<char, int> > cur;
40              for(int q = 0;q < (int)second.avt[j].size();++q) {
41              cur.emplace_back(second.avt[j][q].first , second.avt[j][q].second + size
                    );
42          }
43              result.avt.push_back(cur);
44          }
45          for(int j = 0;j < (int)second.term.size();++j) {  // обновляем терминальные
                вершины
46              result.term.push_back(second.term[j] + size);
47          }
48      }
49      return result;
50  }
51  Automata plus(Automata &first , Automata &second) {
52      auto result = first;
53      int size = (int)first.avt.size();
54      --size;
55      for(int i = 0;i < (int)second.avt[0].size();++i) {  // к корню первого автомата
            подвешиваем детей корня второго автомата
56          result.avt[0].push_back({second.avt[0][i].first , second.avt[0][i].second +
                size});
57      }
58      for(int i = 1;i < (int)second.avt.size();++i) { // добавляем все остальные
            вершины, при этом у каждой обновляем индекс
59              vector<pair<char, int> > cur;
60              for(int j = 0;j < (int)second.avt[i].size();++j) {
61              cur.emplace_back(second.avt[i][j].first , second.avt[i][j].second + size
                    );
62          }
63          result.avt.push_back(cur);
64      }
65      for(int i = 0;i < (int)second.term.size();++i) { // добавляем терминальные
            вершины второго автомата
66          result.term.push_back(second.term[i] + size);
67      }
68      return result;
69  }
70
71  int main() {
72      string expression;
73      cin >> expression;
74      stack <Automata> S;  // стек разбора
75      for(int i = 0;i < expression.length();++i) {
76          if(expression[i] == '+') {
```

```
77            auto second = S.top();
78            S.pop();
79            auto first = S.top();
80            S.pop();
81            auto result = plus(first, second);
82            S.push(result);
83        }
84        else if(expression[i] == '.') {
85            auto second = S.top();
86            S.pop();
87            auto first = S.top();
88            S.pop();
89            auto result = point(first, second);
90            S.push(result);
91        }
92            else if(expression[i] == '*') {
93            auto first = S.top();
94            S.pop();
95            star(first);
96            S.push(first);
97        }
98        else {
99            Automata next(expression[i]);
100           S.push(next);
101       }
102   }
103   auto res = S.top();
104   for(int i = 0;i < (int)res.avt.size();++i) {
105       for(int j = 0;j < (int)res.avt[i].size();++j) {
106           cout << res.avt[i][j].first << ' ' << i << ' ' << res.avt[i][j].second
                  << endl;
107       }
108   }
109   cout << "Terminal\n";
110   for(int i = 0;i < (int)res.term.size();++i) {
111       cout << res.term[i] << ' ';
112   }
113   return 0;
114 }
```

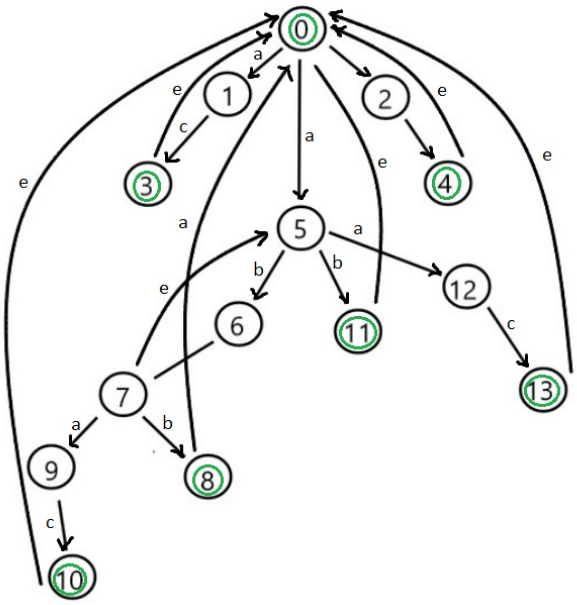
Вывод программы:
Первый пример

```
ab+c.aba.*.bac.+.
a 0 1
b 0 2
a 0 5
c 1 3
c 2 4
  3 0
  4 0
b 5 6
b 5 11
a 5 12
a 6 7
  7 5
b 7 8
a 7 9
  8 0
c 9 10
  10 0
  11 0
c 12 13
  13 0
Terminal
3 4 8 10 11 13 0
```

Второй пример

```
acb..bab.c.*.ab.ba.+.+*a.
a 0 1
b 0 4
a 0 21
c 1 2
b 2 3
  3 0
a 3 16
a 4 5
a 4 12
b 4 14
b 5 6
c 6 7
  7 4
a 7 8
b 7 10
b 8 9
  9 0
a 9 17
a 10 11
  11 0
a 11 18
b 12 13
  13 0
a 13 19
a 14 15
  15 0
a 15 20
Terminal
16 17 18 19 20 21
```

Автомат для первого регулярного выражения:




Автомат для второго регулярного выражения: