

---

Тема 7: Коректност (верификация) на алгоритъм. Аналитична и експериментална проверка на алгоритъм, входни и изходни условия, инвариант на цикъл. Примери за верификация на алгоритъм



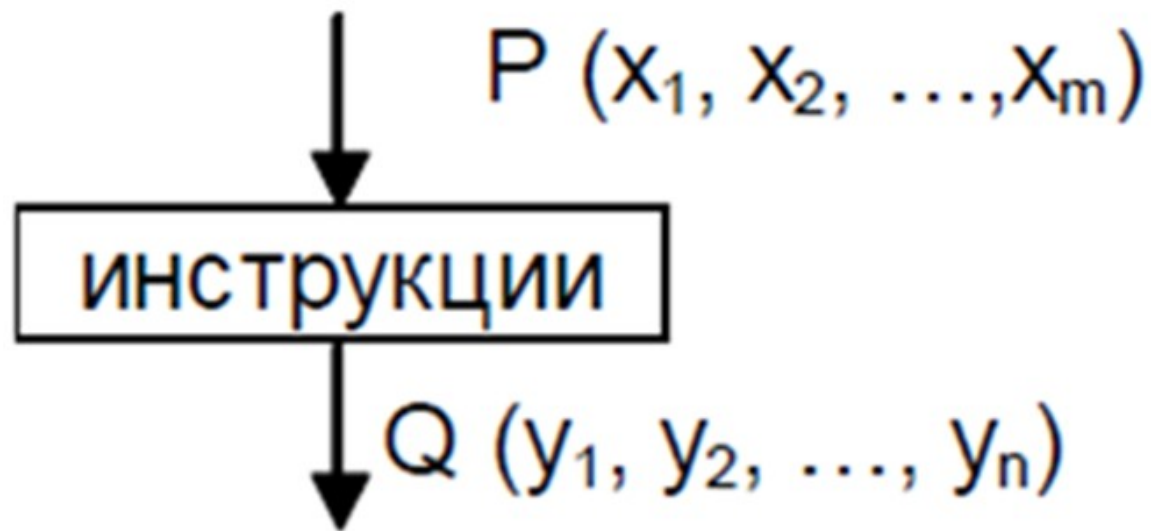
# 1. Начини за проверка на коректността (правилността, верността) на алгоритъм:

а) експериментален: използва се най-често. Използва тестови набори от данни и резултати. По този начин не може да се гарантира 100% коректност на алгоритъма.

б) аналитичен:

- На входа на алгоритъма се задават входни условия **P** (покриващи цялата му дефиниционна област) – това са условия, на които се подчиняват входните променливи на алгоритъма.
- На изхода на алгоритъма се задават изходни условия **Q**, на които се подчиняват резултатите (изходните променливи).
- Във вътрешни точки на алгоритъма също могат да се задават междинни условия.
- Ако при преминаване през инструкциите на целия алгоритъм от входните условия се получат условия, съвпадащи със зададените изходни условия, тогава алгоритъмът е верен.
- Може и обратно – от изходните условия да се търсят входните. Този подход се прилага за не много сложни алгоритми. Както целият алгоритъм, така и всеки негов блок (или даже отделна инструкция) може да се третира по този начин. Входните условия **P** се наричат още **антецеденти**, а изходните условия **Q** – **консеквенти**.



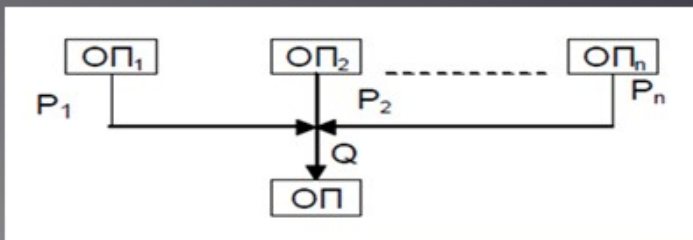


→  $x_1, x_2, \dots, x_m$  – входни променливи,

→  $y_1, y_2, \dots, y_n$  – изходни променливи

## 2. Преобразуване на входните условия в изходни и обратно:

а) при обединение на изходите на няколко оператора:



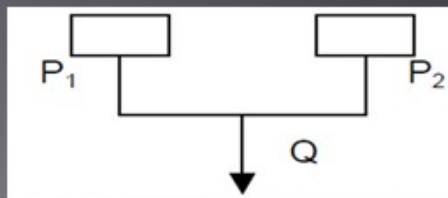
Условието Q се получава като логическо обединение (следствие) на  $P_1, P_2, \dots, P_n$ .

**Пример:**

$P_1: -20 \leq X < 5;$

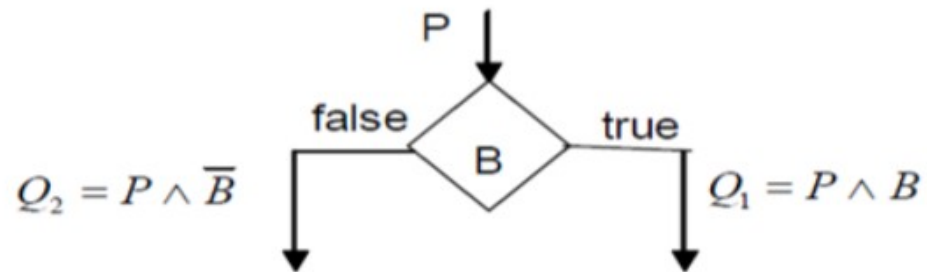
$P_2: 5 \leq X \leq 15;$

От  $P_1 \cup P_2 \Rightarrow Q$  е  $-20 \leq X \leq +15$

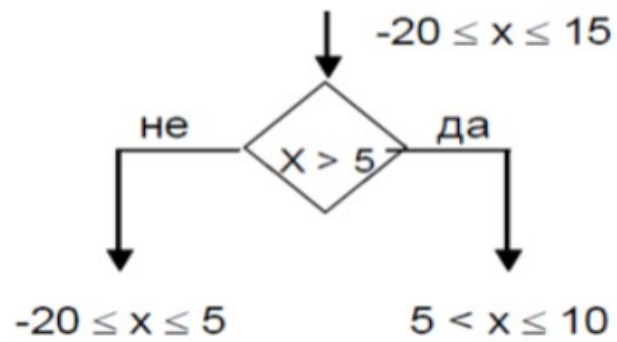


Може да има застъпване на интервалите ( $P_1$  да е същото, а  $P_2$  да е  $2 \leq X \leq 15$ ), но това не променя Q.

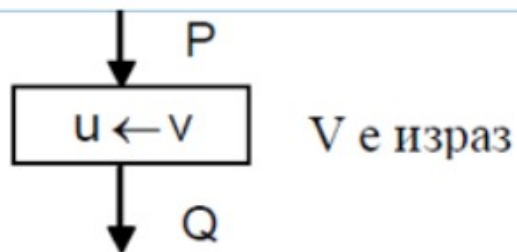
**б) при разклонение на 2 клона:** При входно условие  $P$  и логически израз  $B$  за разклонение имаме изходни условия :  $Q_1 = P \wedge B$  и  $Q_2 = P \wedge \bar{B}$  :



**Пример:**



**в) при оператор за присвояване:**



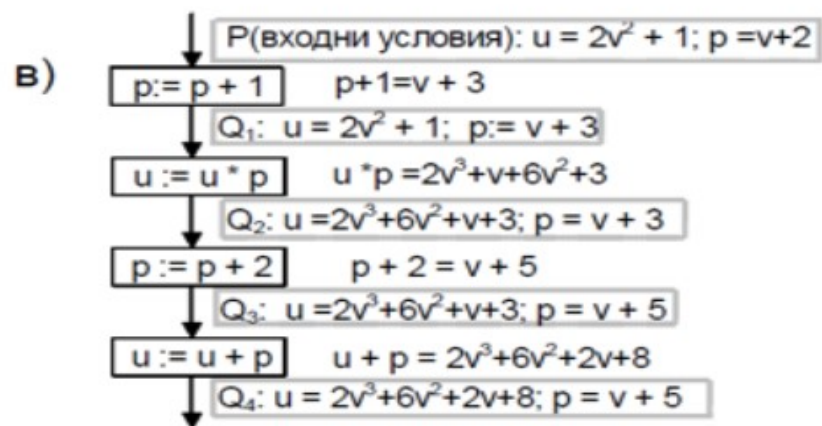
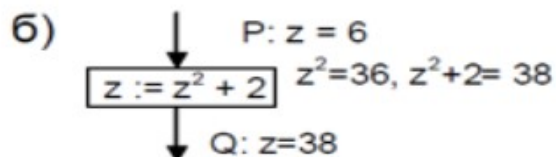
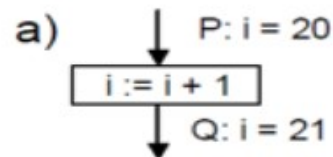
Правилото (наречено ефект на присвояване) е: Нека имаме оператор за присвояване  $u := v$ , където **v** е **израз**, а **u** – **променлива**. Условието **Q** се получава чрез заместване на всички не пресичащи се влягания на израза **v** в **P** с **u**.

Ако в **P** няма нито едно влягане на **u**, тогава **P** се преобразува така, че да има поне 1 влягане.

Обратно, от **Q** може да се получи **P** чрез замяна на всяка поява в **Q** на **u** с **v**.



## Примери:



Чрез прилагане на правилото при присвояване получаваме **Q1 ÷ Q4** при дадени **P** (отгоре-надолу) или обратно (отдолу-нагоре) при дадено **Q4** могат да се получат **Q3, Q2, Q1, P**.

Например, **Q1** се получава от **P** като **P (p = v + 2)** се преобразува така, че в него да има израза  $p + 1$ , т.е.  $p + 1 = v + 3$ , след което  $p + 1$  се заменя с  $p$  и се получава  $p = v + 3$ . Другата част ( $u = 2v^2 + 1$ ) на условие **P** се запазва в **Q1**, тъй като израза  $p + 1$  в него не участва.

г) при оператори за цикъл: най-трудния проблем при доказване на коректността на алгоритъм.

- Нека изобразим цикъла като кръгова последователност от оператори, при която след последния оператор **ОП<sub>n</sub>** на цикъла следва първия ОП<sub>1</sub>.
- Нека в някоя точка (напр. преди **ОП<sub>1</sub>**) на тази последователност запишем условие Р.
- Условието Р ще наречем **инварианта**, ако Р съвпада с **Q<sub>n</sub>** или ако **Р** следва от **Q<sub>n</sub>**. Ако знаем инвариантата, коректността се доказва лесно, но за съжаление инвариантата не може да се определя автоматично - няма правила за определянето ѝ за произволен цикъл.

Очевидно, **при доказване на коректността на алгоритъм определянето на инвариантата на цикъл е ключов проблем – това е творческия елемент в този процес.**



## Примери:

(A) За цикъл *while B do T*; (с тяло T) и при спазване на условия

(а) от входно условие P на T се получава изходно условие P и

(б) от входно условие  $P \wedge B$  на T се получава изходно условие P

следва, че условието в края на цикъла е  $P \wedge \bar{B}$

(B) За цикъл *repeat T until B*; (с тяло T) и при спазване на условия

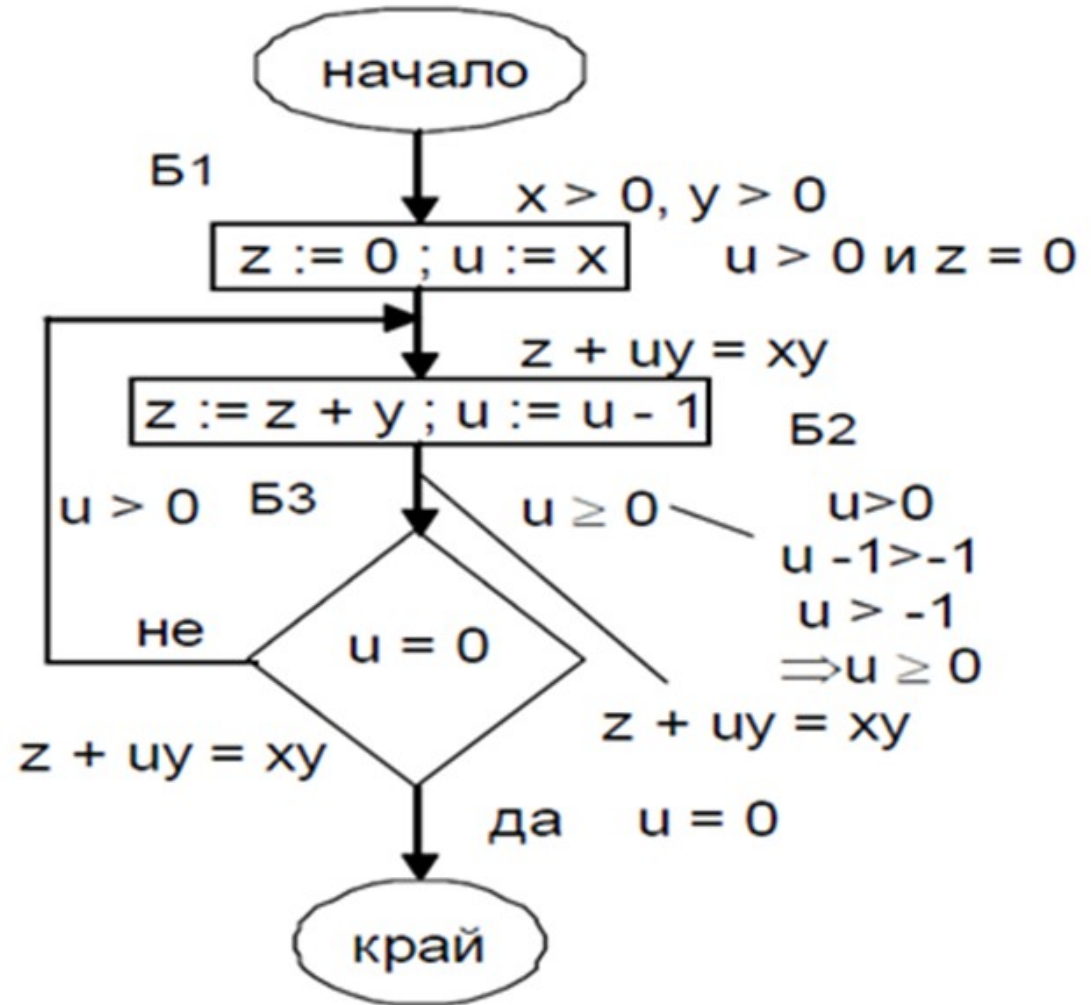
(а) от входно условие P на T се получава изходно условие Q и

(б) от входно условие  $Q \wedge \bar{B}$  на T се получава изходно условие Q

следва, че условието в края на цикъла е  $Q \wedge B$ .

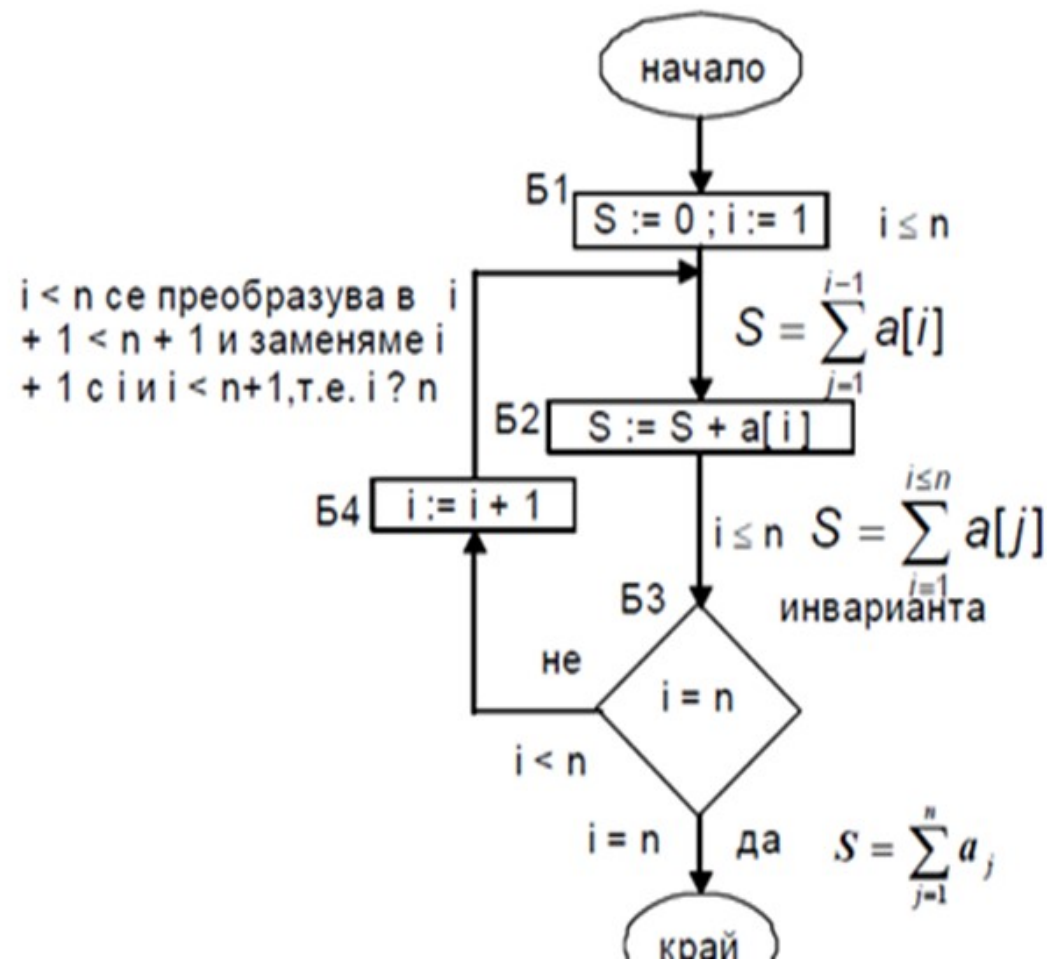
### 3. Примери:

(а) Верификация на алгоритъма  
за умножение чрез събиране:



$$S = \sum_{i=1}^n a[i]$$

(б) Алгоритъм за определяне сумата на елементите на едномерен масив:



# Regular expressions



При много приложения се налага търсене на съответствие на подниз, при наличие на недостатъчна информация за шаблона ( **pattern**), с който се търси съответствие.

- Работещ с текстов редактор, може да иска да създаде **pattern**, съответстващ на различни думи.
- Биолог може да претърсва за геномно съответствие, удовлетворяващо дадени условия.

**Ще представим механизъм, за мощно претърсване на съответствия със сложни М-символни шаблони в N-буквени низови поредици, за време , пропорционално на MN в най-лошия случай.**

*В началото, следва да опишем **шаблона (pattern)**. Такава спецификация трябва да поддържа достатъчно примитиви с по-мощни възможности отколкото операция от вида:*

*“провери дали i-тата буква в текстов низ съответства на j-тата буква на шаблон”.*

Ето защо са необходими **regular expressions**, с чиято помощ се описват шаблони. Да въведем **3 базови операции**.





**Concatenation.** Това е първата операция.

Когато пишем  $AB$ , ние указваме език:  $\{AB\}$  позволяващ 2-буквен низ, формиран чрез сливане на  $A$  и  $B$ .

**Or.** Втората фундаментална операция позволява да специфицираме в шаблона алтернативи.

Ако поддържаме през **or** 2 алтернативи, то и двете следва да са допустими в езика.

Ползваме символа  $|$  за да отбележим тази операция.

Например,  $A|B$  специфицира език  $\{A, B\}$  и

$A|E|I|O|U$  специфицира език:  $\{A, E, I, O, U\}$ .

Операцията **concatenation** има по-висок приоритет от **or**, така че

$AB|BCD$

специфицира език

$\{AB, BCD\}$ .

**Closure.** Третата фундаментална операция позволява части от шаблона да бъдат **повторяеми** в произволен ред.

Отбелязвам **closure** чрез поставяне на  $*$  след шаблона, който ще се повтаря.

**Closure** имат по-висок приоритет от **concatenation**,

Така че  $AB^*$  указва езикова възможност за формиране на низ с  $A$ , следвана от 0

Или повече  $B$ -та, докато  $A * B$  специфицира езикова възможност за низове с 0

или повече  $A$ -та, следвани от  $B$ .

**Parentheses.** Използваме скоби за променяне на подразбиращата се последователност. Например,

**$C(AC|B)D$**

указва езикова възможност за формиране на  $\{CACD, CBD\}$ ;

**$(A|C)((B|C)D)$**

указва езикова възможност за формиране на  $\{ABD, CBD, ACD, CCD\}$ ; и

**$(AB)^*$**  указва езикова възможност за формиране на низове чрез сливане на произволен брой поднизове  $AB$ , включително и липса на такива:  
 $\{, AB, ABAB, \dots\}$ .

**RE**

$(A|B)(C|D)$

$A(B|C)^*D$

$A^* | (A^*BA^*BA^*)^*$

$BABBA^*A^*$

**съответствия**

$AC AD BC BD$

$AD ABD ACD ABCCBD$

$AAA BBAABB BABAAA$

**несъответствия**

every other string

$BCD ADD ABCBC$

$ABA BBB$

*примери с **regular expressions***

## Дефиниция

**regular expression (RE)** е или

- празна поредица
- една буква
- **regular expression** включен в скоби
- 2 или повече сляти заедно **regular expressions**
- 2 или повече **regular expressions**, разделени с оператор **or (|)**
- **regular expression**, следвана от **closure operator (\*)**

## Описания на символи.

символът **(.)** обозначава „коя да е“ буква. Поредица букви обградени в **[] square brackets** обозначава „коя да е“ от тези букви.

Това може да се нарече и „изброимо множество на допустимите символи“.

Ако се предхожда от **^**, поредица, заградена в **[]**, то указваме „коя да е, но поне една, от тези букви“.

име	знак	пример
Wildcard	.	A.B
Набор, заграден в <b>[]</b>		<b>[AEIOU]*</b>
множество заградено в <b>[]</b>		<b>[A-Z]</b>
и отделено с <b>-</b>		<b>[0-9]</b>
заградено в <b>[]</b>		
и предхождано от <b>^</b>		<b>[^AEIOU]*</b>

описания на служебни символи

## Символи, свързани с повторения ( Closure )

**closure operator** указва някакъв брой повторения на копия от операнда му:

знак **+** указва поне 1 копие,      знак **?** указва 0 или 1 копие и

**число или range**, указани в скоби **{ }** специфицират брой повторения.


## Escape последователности

Някои символи, като **\**, **.**, **|**, **\***, **(**, and **)** са *metacharacters*,

които ползваме за формиране на **regular expressions**.

Escape последователност, започваща с **\** служи за разделяне на **metacharacters** от нормални символи на азбуката.

escape последователност може да е **\**, следвана от един **metacharacter** (което е представа за тази буква). Например, **\\** представя **\**. **\t** представя **tab character**, **\n** представя нов ред и **\s** представя празен символ.

опции	notation	пример	значение	от език	недопустим
Поне едно	<b>+</b>	<b>(AB)+</b>	<b>(AB)(AB)*</b>	AB ABABAB	 empty <b>€</b> BBBAAA
0 или 1	<b>?</b>	<b>(AB)?</b>	<b>€   AB</b>	<b>€</b> AB	any other string
Указан в { } брой		<b>(AB){3}</b>	<b>(AB)(AB)(AB)</b>	ABABAB	any other string
От-до в { }		<b>(AB){1-2}</b>	<b>(AB)   (AB)(AB)</b>	AB ABAB	any other string

знаци за **Closure** (указващи броя повторения)





## RE в приложения

### - Търсене на подниз

Целта е разработване на алгоритъм , който да определи дали даден низ Принадлежи на допустимите низове, съобразно дадено описание чрез **regular expression**.

### - Валидация

Често **RE** служат за валидиране на входни данни .

Примери:

#### context

Търсене на подниз  
phone number  
Java identifier  
Pattern\_Matcher  
genome маркер  
email адрес

#### regular expression

**\*NEEDLE.\***  
**\([0-9]{3}\)\ [0-9]{3}-[0-9]{4}**  
**[\$\_A-Za-z][\$\_A-Za-z0-9]\***  
  
**gcg(cgg | agg)\*ctg**  
**[a-z]+@([a-z]+\.)+(edu | com)**

#### допуснат низ

**A HAYSTACK NEEDLE IN**  
**(800) 867-5309**  
  
**gcgaggaggcggcggctg**  
**[rs@cs.princeton.edu](mailto:rs@cs.princeton.edu)**

типични **regular expressions** в приложения





## Недетерминиран краен автомат (**Nondeterministic Finite-state Automata – NFA**)

Припомнете си алгоритъма **Knuth- Morris-Pratt** , който е типичен представител на крайните автомати и служи да сканира текст за съвпадение с шаблон (**search pattern**). Крайният автомат **КМП** преминаваше от състояние в състояние, като преглежда следващ символ.

Автоматът регистрира съвпадение, само ако достигне крайно състояние - **accept**. Такъв автомат е **детерминиран**: всеки преход в състояние е детерминиран от следващия символ в текста.

*За да работим с **regular expressions**, се нуждаем от по-съвършена машина:*

Автоматът не е в състояние да определи дали шаблонът съвпада в даден момент, тествайки единствено един символ – как да се следят повторенията (**closures**). Не може да се определи и колко символа (групи) следва да се проверят докато се съобщи за съвпадение или несъвпадение.

Въвеждаме недетерминираност (**nondeterminism**): това са случаите, в които съществува повече от 1 вариант за проверка на съвпадение с шаблон и автоматът трябва да достигне до правилния.

Ще покажем, че е лесно създаването на програма, изпълняваща **недетерминиран краен автомат (NFA)** , както и ще покажем симулация на операциите му.

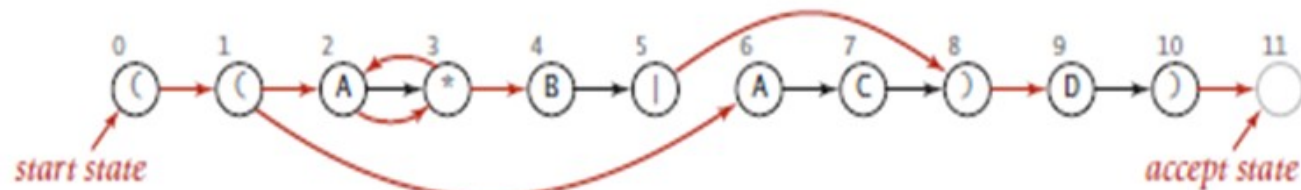
**Теоремата на Клайн**, едно от фундаменталните твърдения в компютърната наука, постановява, че винаги съществува **NFA** съответстващ на всеки **RE** (както и обратното).

$$\mathbf{NFA} \leftarrow \rightarrow \mathbf{RE}$$

Ще покажем трансформацията от **RE** към **NFA** и ще проследим операциите на **NFA** за изпълнение на задачата.

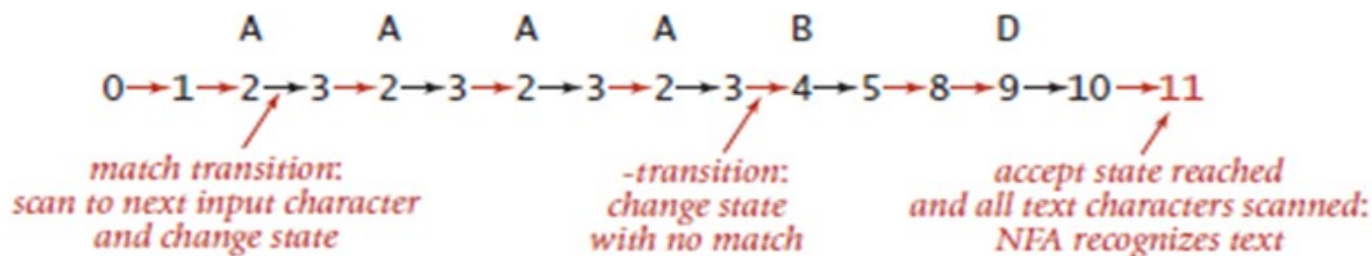
Преди да проследим как да изградим **NFA**, съответстващ на определен шаблон (**RE**), ще дадем пример, илюстриращ базовите операции. На следващата фигура е показан **NFA** който установява дали низ принадлежи на езика, описан с **RE**:

$$((\mathbf{A} * \mathbf{B} \mid \mathbf{AC})\mathbf{D})$$

**NFA** съответстващ на шаблон **( ( A \* B | A C ) D )**

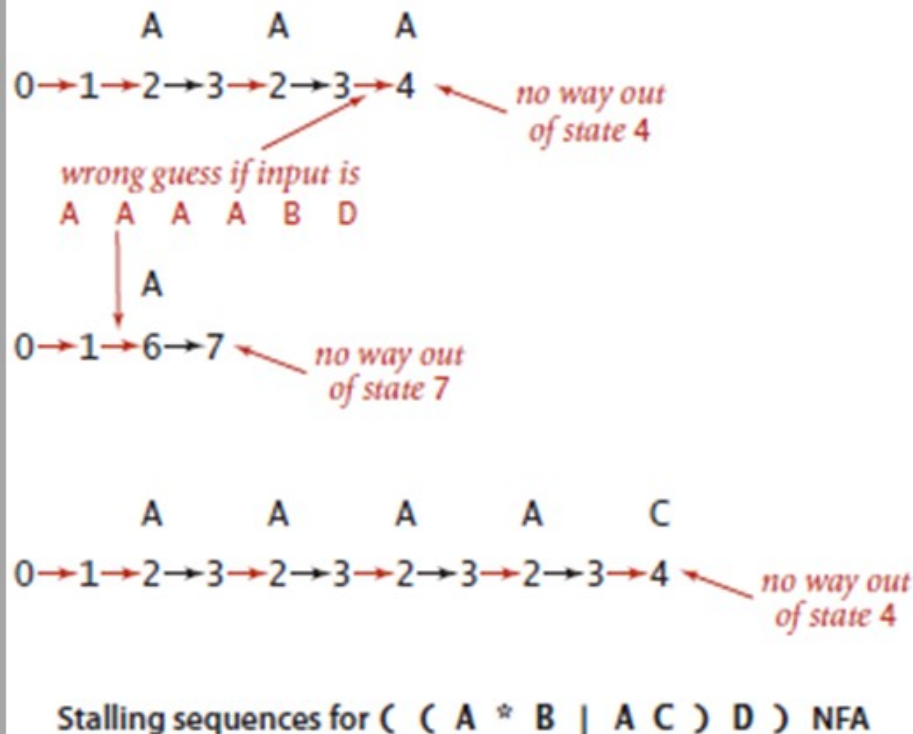
Входен низ: AAAABD:



Път на намиране на съответствие с **( ( A \* B | A C ) D )** NFA

Указано е значението на :





Примерът илюстрира, че са възможни последователности, каращи автомат да не достигне изход, дори и при прост текст като **A A A A B D** (който е разпознаваем).

Например, ако автомата прави преход към състоянието 4, преди да сканира всички **A**-та, попада в състояние без изход, тъй като единственият преход от състояние 4 навън е откриване на **B**.

Или отиде в 6, а липсва символ C.

Горните 2 примера онагледяват необходимостта от недетерминиран подход.

виждаме основната разлика между **NFA** и **DFA**:

тъй като **NFA** може да има множество изходящи връзки от всяко състояние, преходът навън от състояние е недетерминиран — възможни са различни преходи, при различен брой сканирани символа.



## Софтуерна представа за поведението на NFA

### представяне

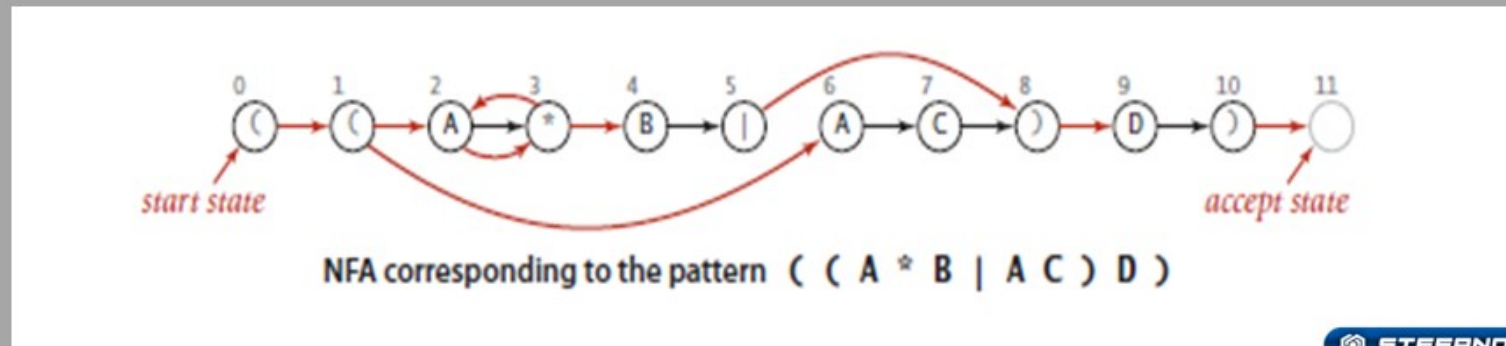
**RE** поражда **поредица от състояния** в които попада автомата ( цели числа, от 0 до **M**, където **M** е броя на символите в израза - **RE**).

Т.е. ще строим граф – според **RE**, числените стойности на състоянията му съответстват на масив (**re[]**) от **char**. Така ще отразим допустимите преходи в състояния – ако **re[i]** съвпада с елемент от азбуката на езика, имаме допустим преход от **i** към **i+1**).

Представянето на **(empty)-преходите** е през т.нар. **digraph** — тези преходи се представят с връзки ( **означените с червено** в диаграмата на графа). Всички допустими преходи са в графа.

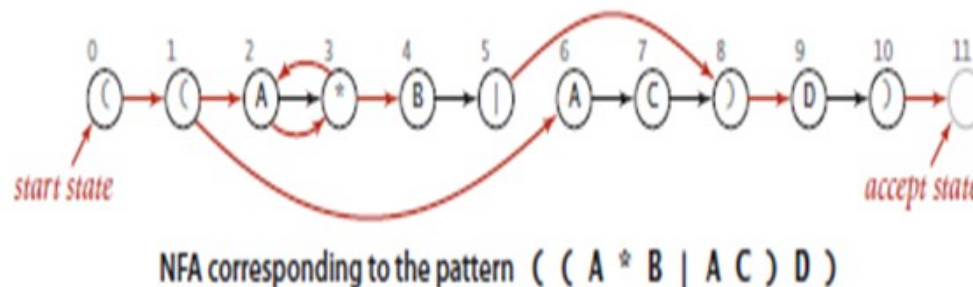
**Empty – преходите (€)** съставят т.нар. **digraph G**.  
в **digraph** за нашия пример ще имаме 9 възела:

**0 → 1    1 → 2    1 → 6    2 → 3    3 → 2    3 → 4    5 → 8    8 → 9    10 → 11**





**NFA – симулация на поведението и достижимост.**



За правилната симулация на *NFA*, наизм всички преходи в състояния, които са се срещнали в процеса на действие на автомата по изследване текущ символ от входния поток – т. нар. изследване достижимост при множествен вход (*multiple-source reachability computation*).

Например, възможен набор от състояния за нашия *NFA* може да е : 0 1 2 3 4 6. Ако първия символ е **A**, *NFA* автоматът може да има **match transition** към 3 или 7. След това може да премине чрез **ε-transitions** от 3 към 2 или от 3 към 4. Така че, множеството допустими състояния, водещи до **match transition** за втория символ са 2 3 4 и 7.

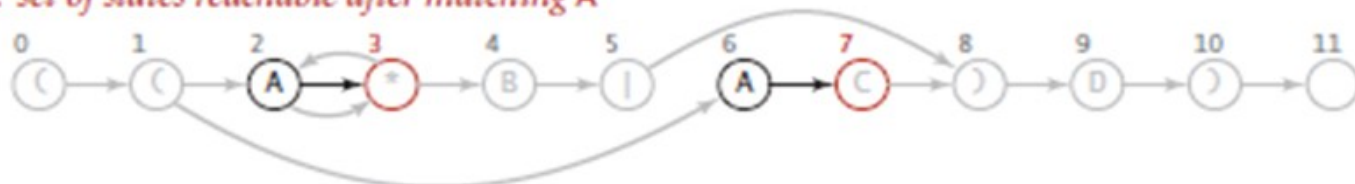
Итерирайки в този процес, достигае до момента в който всички входни символи са се изчерпали – и това води до един от двата резултата:

- достигнали сме до крайния - **accept state**.
- множеството от достижими състояния не включва **accept state**.

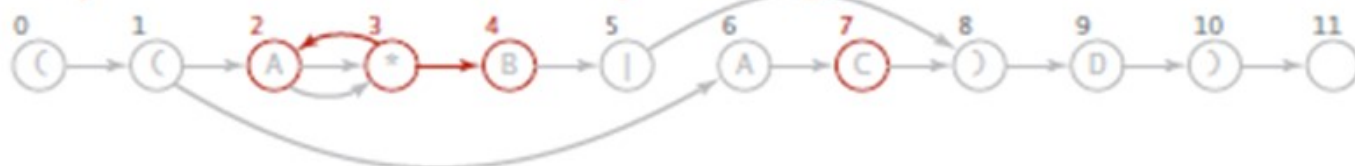
0 1 2 3 4 6 : set of states reachable via  $\epsilon$ -transitions from start



3 7 : set of states reachable after matching A



2 3 4 7 : set of states reachable via  $\epsilon$ -transitions after matching A



3 : set of states reachable after matching A A

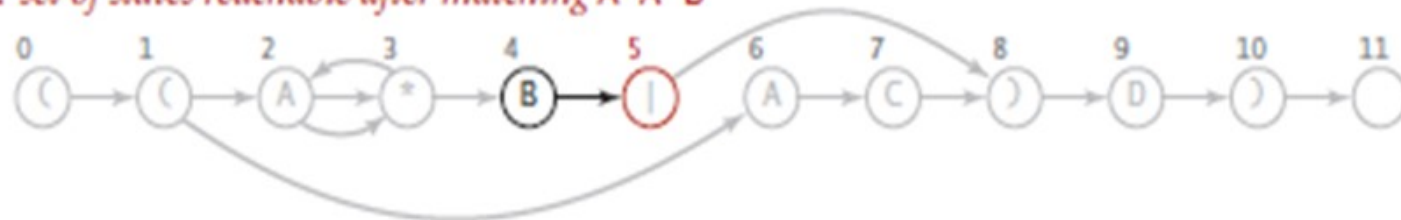


2 3 4 : set of states reachable via  $\epsilon$ -transitions after matching A A

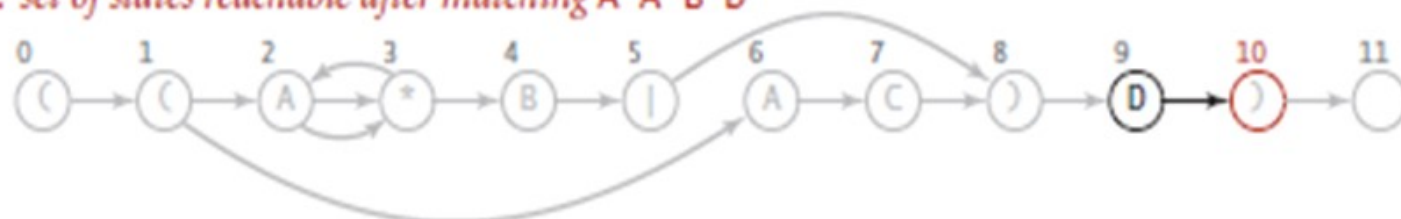
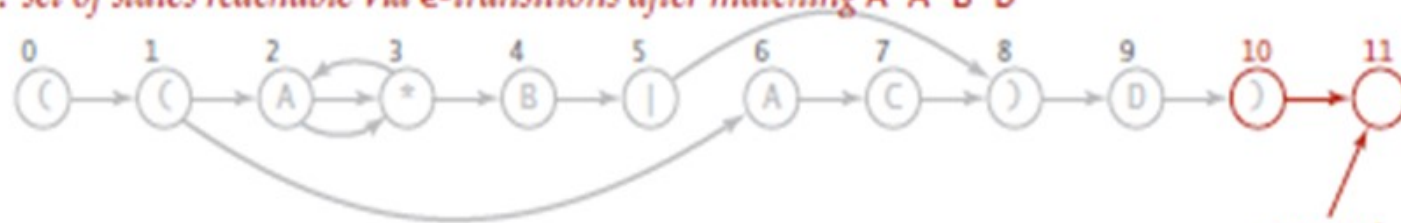


2/2

5 : set of states reachable after matching A A B

5 8 9 : set of states reachable via  $\epsilon$ -transitions after matching A A B

10 : set of states reachable after matching A A B D

10 11 : set of states reachable via  $\epsilon$ -transitions after matching A A B D

accept !

Възможни преходи при симулацията за вход **AABD** при **NFA** автомат за **RE**

( ( A \* B | A C ) D )

## Изграждане на NFA, съответстващ на даден RE

Да построим **digraph G**, който се състои от всички **€-transitions**.

Ще използваме и стек за да проследим позиционното съответствие за срещнатите „(„ и „or“ операции.

### Отработване на операция **Concatenation**.

С помощта на NFA, операцията – **concatenation** е една от най-лесните за имплементация. Допустимите преходи (Match transitions) за състояния, кореспондиращи на символите в азбуката на конкретния език (описан с RE), всъщност неявно реализират тази операция.

### Обработка на скоби.

Вкарваме в стек RE индекса за всяка срещната „(„. Когато срещнем съответстващата й „)“, извличаме от стека съответстващата лява скоба, по начин показан по-долу.

### Обработване на операция - **Closure**.

Операция - **closure** (\*) може да се срещне :

- след единичен символ, когато добавяме **€-transitions** към и от символа ;
- или
- след дясна скоба „)“, когато ще добавим **€-transitions** to и from съответстващата лява скоба и то стоящата на върха на стека.





## Обработка на Or expression.

обработваме RE от вида  $(A \mid B)$ , където  $A$  и  $B$  са елементи на RE езика, с добавяне на 2  $\epsilon$ -transitions:

- Един от възела за „(“ към възела за обработка на първия символ от частта „B“, и
- Един от възела, съответстващ на оператора „|“ към възела, съответстващ на „)“.

В стек поставяме RE index, съответстващ на операцията „|“ (така също и за индекса, съответстващ на лявата скоба, описана по-горе). Така информацията, която ни е нужна е на върха на стека при необходимост – а това е в момента, когато достигнем дясна скоба.

Така описаните  $\epsilon$ -transitions позволяват NFA да избере една от двете алтернативи. Не следва да добавяме  $\epsilon$ -transition от възела съответстващ на оператора „|“ към възел за състояние с по-висок индекс, така както правим при всички други ситуации. Единственият начин, в който NFA излиза успешно от това състояние е преход към състояние, съответстващо на „)“.



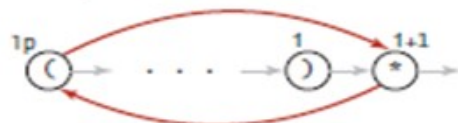
## граф и код за описаните ситуации

single-character closure



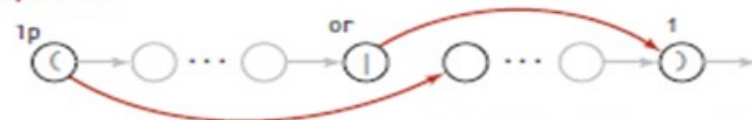
```
G.addEdge(i, i+1);  
G.addEdge(i+1, i);
```

closure expression



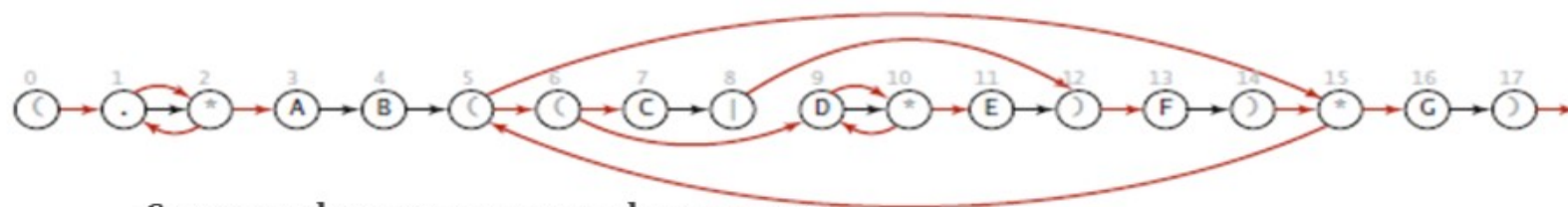
```
G.addEdge(1p, i+1);  
G.addEdge(i+1, 1p);
```

or expression



```
G.addEdge(1p, or+1);  
G.addEdge(or, i);
```

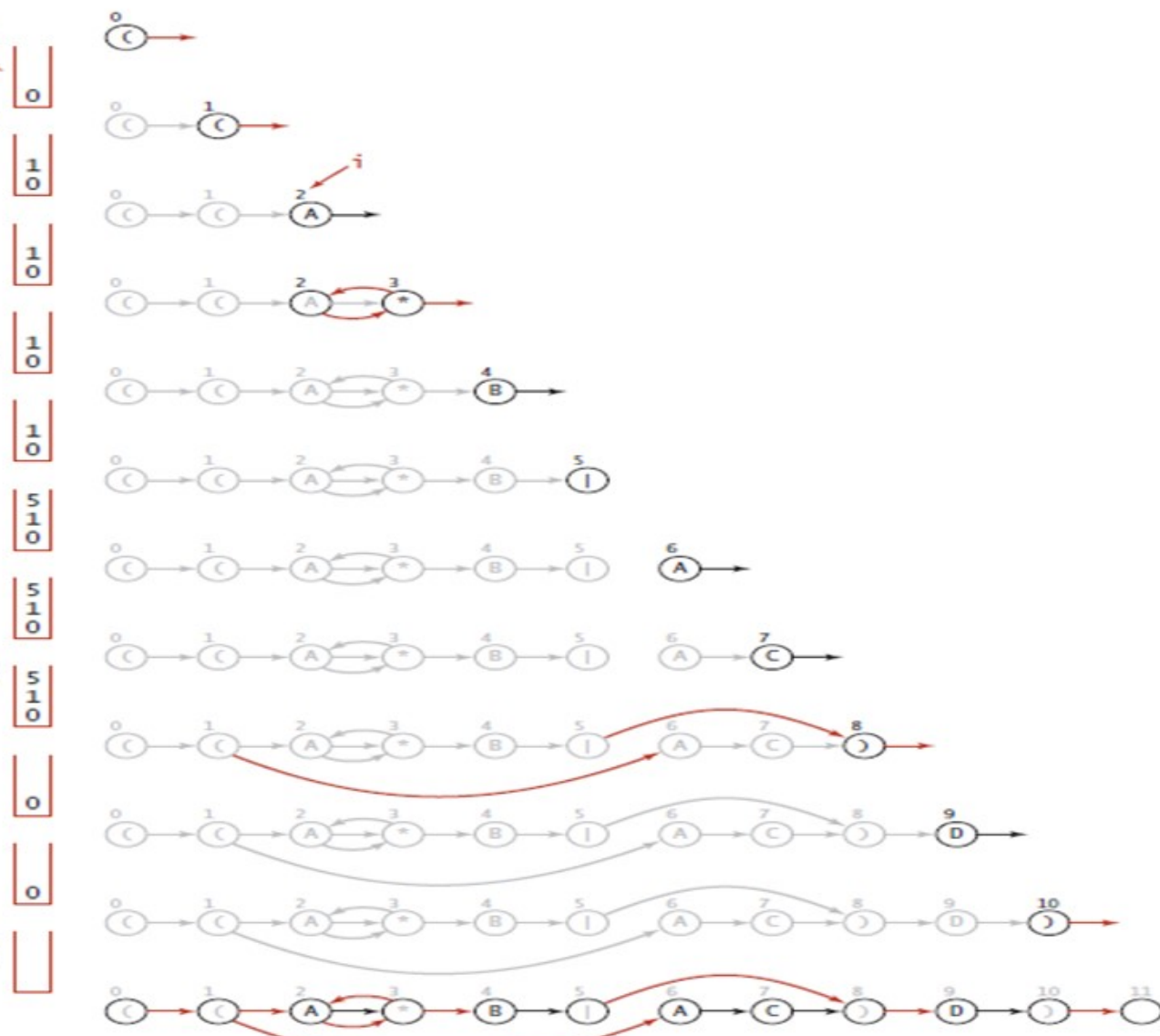
NFA construction rules



С таква формализми сме формирали  
NFA за обработка на RE израза:  $(. * A B (( C | D * E ) F ) * G )$

stack for  
indices of  
left parentheses  
and ops  
(ops)

Стек:  
вкарваме  
(  
и  
|



Запълване на стека на NFA за RE:

$( ( A * B | A C ) D )$

Кодът,  
който  
конструира  
графа от  
предходната  
страница

Конструкторът на  
класа **NFA** строи  
**NFA** графа,  
съответстващ на  
предния **RE** израз.

Всъщност се строи  
digraf на  
€- преходите

```
public class NFA
{
    private char[] re;           // match transitions
    private Digraph G;          // epsilon transitions
    private int M;               // number of states

    public NFA(String regexp)
    { // Create the NFA for the given regular expression.
        Stack<Integer> ops = new Stack<Integer>();
        re = regexp.toCharArray();
        M = re.length;
        G = new Digraph(M+1);

        for (int i = 0; i < M; i++)
        {
            int lp = i;
            if (re[i] == '(' || re[i] == '|')
                ops.push(i);
            else if (re[i] == ')')
            {
                int or = ops.pop();
                if (re[or] == '|')
                {
                    lp = ops.pop();
                    G.addEdge(lp, or+1);
                    G.addEdge(or, i);
                }
                else lp = or;
            }
            if (i < M-1 && re[i+1] == '*') // lookahead
            {
                G.addEdge(lp, i+1);
                G.addEdge(i+1, lp);
            }
            if (re[i] == '(' || re[i] == '*' || re[i] == ')')
                G.addEdge(i, i+1);
        }
    }

    public boolean recognizes(String txt)
    // Does the NFA recognize txt?
}
```

Кодът на метод `recognize(String)` чрез който **NFA** който проверява за съответствие с `pattern` (това е израз **RE**).

Използва прост алгоритъм за търсене на съответствие на подниз.

```
public boolean recognizes(String txt)
{ // Does the NFA recognize txt?
  Bag<Integer> pc = new Bag<Integer>();
  DirectedDFS dfs = new DirectedDFS(G, 0);
  for (int v = 0; v < G.V(); v++)
    if (dfs.marked(v)) pc.add(v);

  for (int i = 0; i < txt.length(); i++)
  { // Compute possible NFA states for txt[i+1].
    Bag<Integer> match = new Bag<Integer>();
    for (int v : pc)
      if (v < M)
        if (re[v] == txt.charAt(i) || re[v] == '.')
          match.add(v+1);
    pc = new Bag<Integer>();
    dfs = new DirectedDFS(G, match);
    for (int v = 0; v < G.V(); v++)
      if (dfs.marked(v)) pc.add(v);
  }

  for (int v : pc) if (v == M) return true;
  return false;
}
```



И викащата предните методи програма:

1. Конструира графа **NFA** за **RE** – **regexp**;
2. Проверява за съответствие на входен низ- **txt** с правилата на **RE**;

## GREP

```
public class GREP
{
    public static void main(String[] args)
    {
        String regexp = "(.*" + args[0] + ".*)";
        NFA nfa = new NFA(regexp);
        while (StdIn.hasNextLine())
        {
            String txt = StdIn.hasNextLine();
            if (nfa.recognizes(txt))
                StdOut.println(txt);
        }
    }
}
```