



# *Методи на Транслация*

Лексикален анализ.

Синтактичен анализ.

Управление на грешки.

# Азбука



# Азбука

## Определение:

Множеството от знаци, допустими в един ЕП наричаме азбука.

Азбуката обикновено се дели на непресичащи се под множества, като букви, цифри и др.

'a', 'b', 'c', 'd', 'e', ... 'z'

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

# *Лексикален Анализ*

## *(scanner)*



## Лексика

### Определение:

Лексемите на един ЕП са по-големи синтактични елементи съставени от един или повече знаци на азбуката на ЕП.

Обикновено лексиката се описва с регулярна граматика.

### Видове:

- ❖ Идентификатори – например ***abc1***;
- ❖ Ключови думи – ***while, if, for, ...***;
- ❖ Резервирани думи – запазена за в бъдеще кл. дума ***task***;
- ❖ Шумови думи – например ***goto, go to, go; to*** е такава защото ***go[to]***;
- ❖ Коментари, Разделители;
- ❖ Оператори и Ограничители/Разделители и др.;



# Регулярни граматики

## Определение:

Една граматика е регулярна, когато може да се опише с правила от вида:

$$A = a \quad a, b \in \text{TC}$$

$$A = b B \quad A, B \in \text{HTC}$$

## Пример: Граматика за идентификаторите

Ident = Letter | Letter Other.

Other = Letter | Digit | Letter Other | Digit Other

Letter = 'a'..'z' | 'A'..'Z' | '\_' | '.'.

Digit = '0'..'9'.

Например извода за *tst1* е:

*Ident*  $\Rightarrow$  *Letter Other*  $\Rightarrow$  *Letter Letter Other*  $\Rightarrow$  *Letter Letter Letter Other*  $\Rightarrow$  *Letter Letter Letter Digit*



# Регулярни граматики

## Друго определение:

Една граматика е регулярна, когато може да се опише с не рекурсивна EBNF

**Пример:** *Грамматика за идентификаторите*

```
Ident = Letter {Letter | Digit}.
```

# Детерминиран краен автомат (ДКА)

Могат да бъдат използвани за анализ на регулярни езици  
(езици породени от регулярни граматика)

## Определение:

Детерминиран краен автомат е наредена петорка  $(S, I, \delta, s_0, F)$ , където:

- ❖  $S$  – вътрешна азбука (множество състояния);
- ❖  $I$  – външна азбука (множество от входни символи);
- ❖  $\delta: S \times I \rightarrow S$  – функция на преход;
- ❖  $s_0$  – начално състояние;
- ❖  $F$  – множество от крайни състояния;

Следователно, езикът, който се разпознава от ДКА е множеството от всички последователни символи, които водят от началното състояние до някое от крайните състояния.



# Пример за ДКА

Пример:

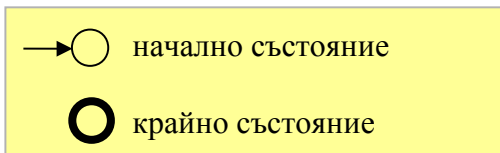
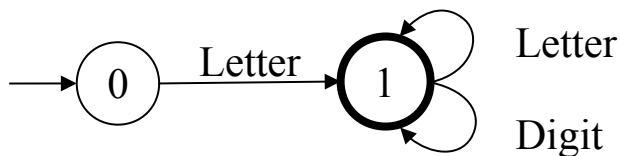


Таблица на състоянията

$\delta$	Letter	Digit
s0	s1	грешка
s1	s1	s1

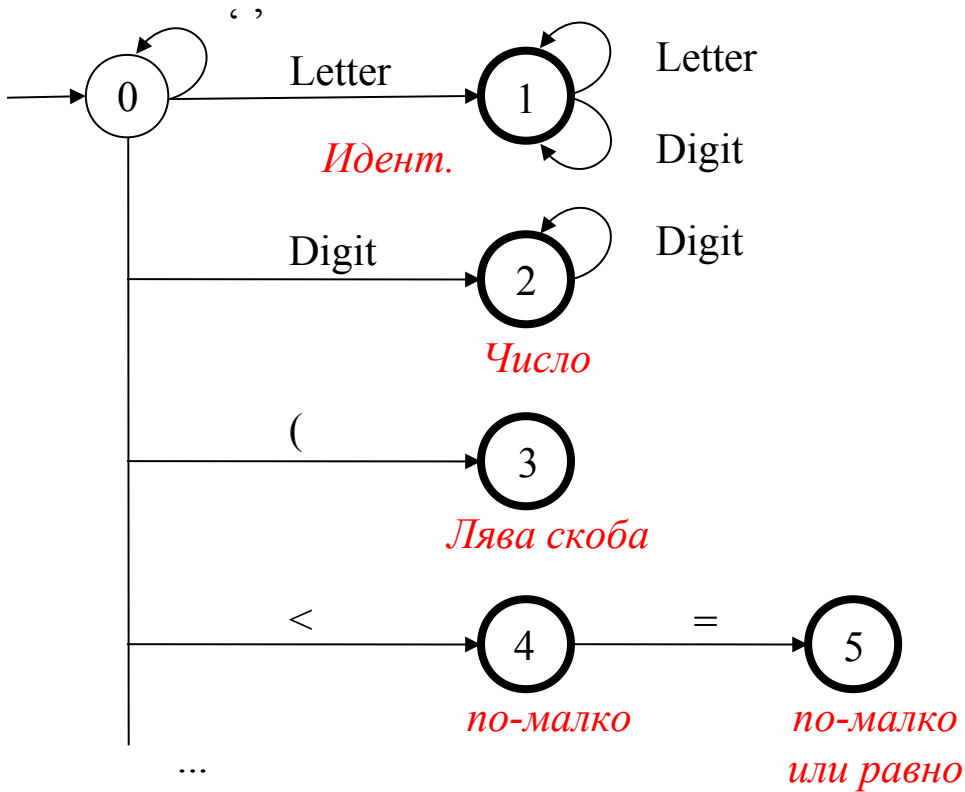
“краен”, защото  $\delta$  може да бъде записана явно

ДКА разпознава изречение:

- ❖ ако е в някое крайно състояние;
- ❖ и ако входните символи са напълни консумирани или няма възможен преход със следващия символ;

# Скенера̀т като ДКА

Скенера̀т може да бъде разглеждан като един голям ДКА.



Примерен вход: test <= 3

$s_0 \xrightarrow{\text{test}} s_1$

- ❖ Няма преход при ' ' към  $s_1$
- ❖ Лексемата "Идент." е разпозната

$s_0 \xrightarrow{<=} s_5$

- ❖ Прескача интервалите в началото
- ❖ Не спира в  $s_4$
- ❖ Няма преход при ' ' в  $s_5$
- ❖ Лексемата "по-малко" е разпознава

$s_0 \xrightarrow{3} s_2$

- ❖ Прескача интервалите в началото
- ❖ Няма преход при ' ' в  $s_2$
- ❖ Лексемата "Число" е разпознава

След всяка разпозната лексема скенера започва от  $s_0$  отново!



# Действие на ДКА при анализ на входен поток символи

i	f		(	x	=	=	3	)	\n	
---	---	--	---	---	---	---	---	---	----	--



При всяко влизане в крайно състояние (без възможност да се приложи нов преход към друго състояние) ДКА спира и връща типа на откритата лексема, както и нейната стойност (низ).

Keyword	SpecialSymbol	Ident	SpecialSymbol	Number	SpecialSymbol
<b>if</b>	<b>(</b>	<b>x</b>	<b>==</b>	<b>3</b>	<b>)</b>

# *Синтактичен Анализ*

## *(parser)*



# Синтаксис

## Определение:

Синтаксис на един ЕП се наричат правилата, на които той трябва да отговаря за да можем да кажем, че програмата е правилно форматирана т.е. с правилна структура. Синтаксиса на ЕП се определя с граматика.

## Синтактични елементи на ЕП:

- ❖ Операции: + – \*
- ❖ Изрази:  $(a + b) * c$ ;
- ❖ Декларации: *int x*;
- ❖ Прости оператори: присвояване, за В/И, безусловен преход, активиране/извикване на подпрограма и др.;
- ❖ Съставни оператори;

# Контекстно-свободни граматики

## Проблем

Регулярните граматики не могат да се справят с централната рекурсия

$$E = x \mid '(E)'$$

За подобни случаи са необходими контекстно-свободни граматики.



# Контекстно-свободни граматики

## Определение

Една граматика е контекстно-свободна граматика (КСГ), ако всички нейни произведения са от вида:

$A = \alpha$ .  $A \in \text{НТС}$ ,  $\alpha$  е непразна последователност от ТС и НТС

В EBNF дясната страна може да съдържа и мета-символите  
 $|$ ,  $()$ ,  $[]$ , и  $\{\}$

## Пример:

Expr = Term {'+' Term}.

Term = Factor {'\*' Factor}.

Factor = id | '(' Expr ')'. ←

Индиректна централна рекурсия

Контекстно-свободните граматики се разпознават от **стекови автомати**

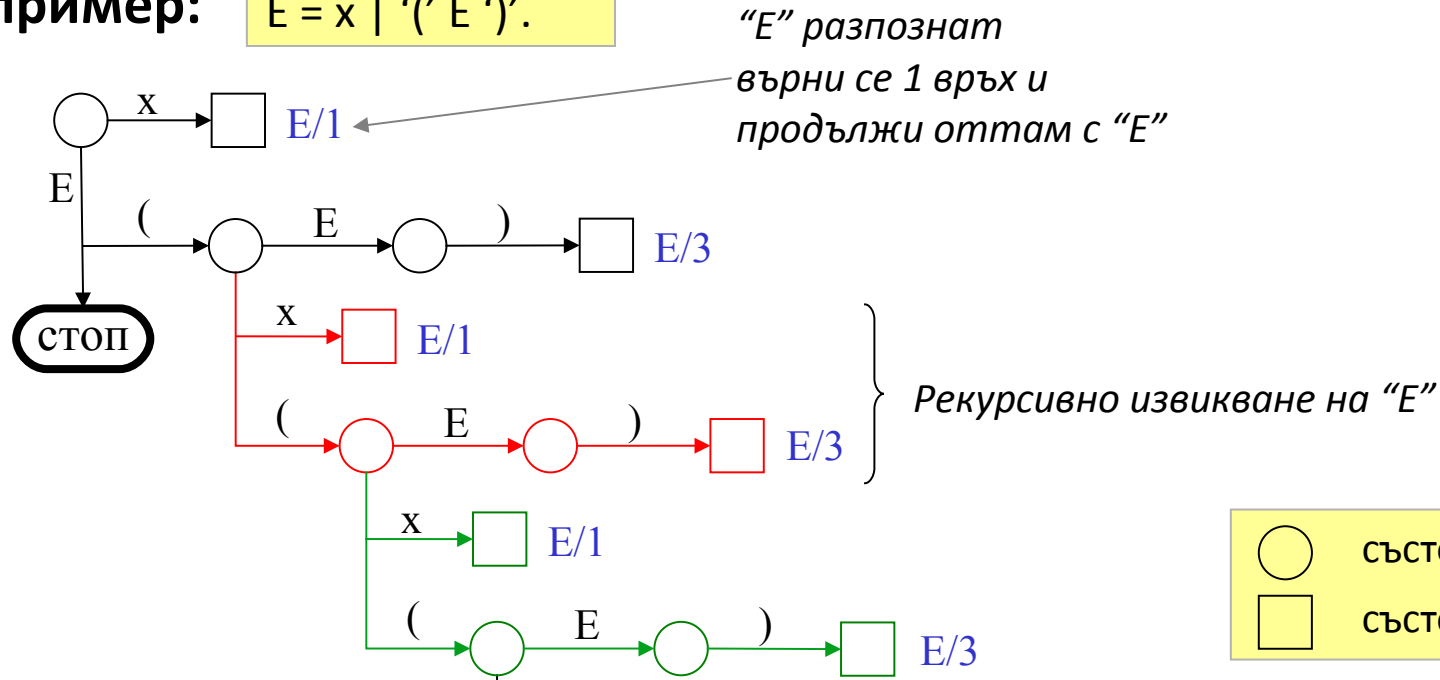


# Стекови автомати

## Характеристики:

- ❖ Позволява преходи с терминални и нетерминални символи;
- ❖ Използва стек за да запомни състоянията, през които е преминал;

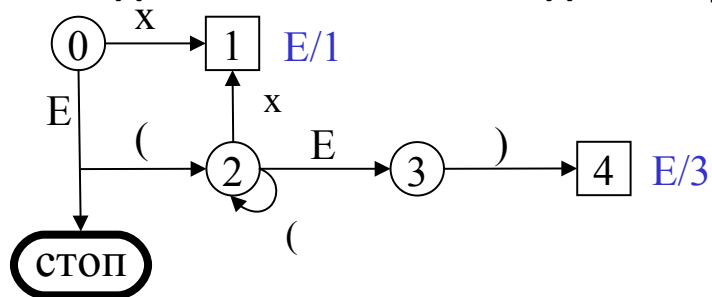
Например:  $E = x \mid (' E ')$ .





# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

Преминатите състояния се съхраняват в стек

Стек

0

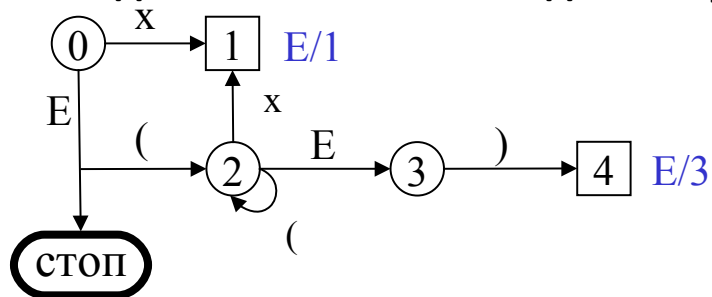
Оставащ вход

(( x ))

0

# Действие на стеков автомат

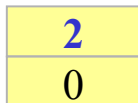
Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

Преминатите състояния се съхраняват в стек

Стек



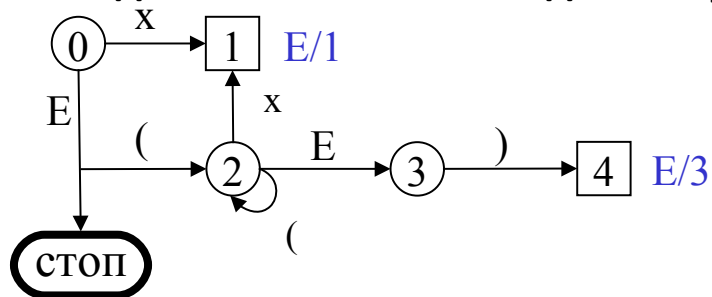
Оставащ вход

**(( x ))**



# Действие на стеков автомат

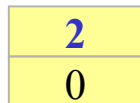
Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

Преминатите състояния се съхраняват в стек

Стек



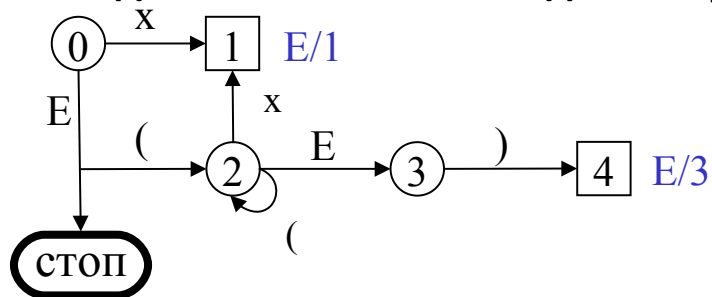
Оставащ вход

**(( x ))**



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

Преминатите състояния се съхраняват в стек

Стек

2
2
0

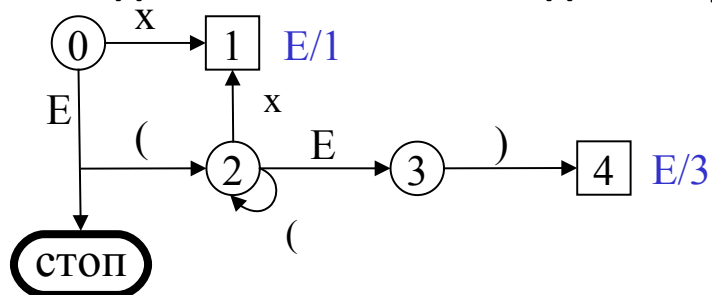
Оставащ вход

**( x )**



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

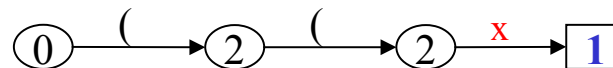
Преминатите състояния се съхраняват в стек

Стек

1
2
2
0

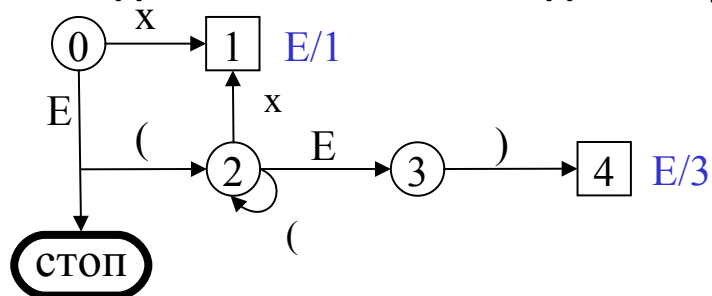
Оставащ вход

x ))



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

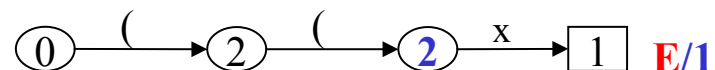
Преминатите състояния се съхраняват в стек

Стек

2
2
0

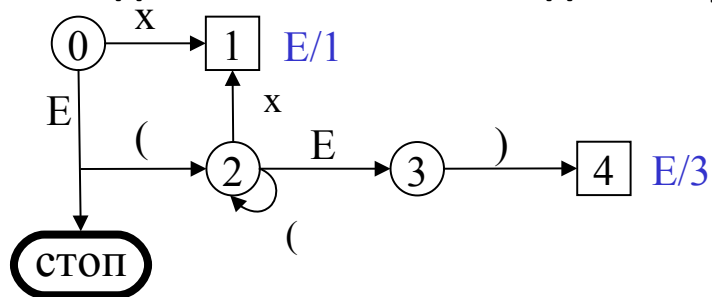
Оставащ вход

**E** ))



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

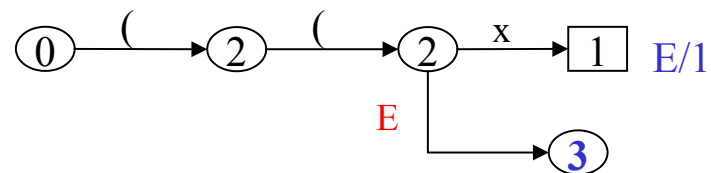
Преминатите състояния се съхраняват в стек

Стек

3
2
2
0

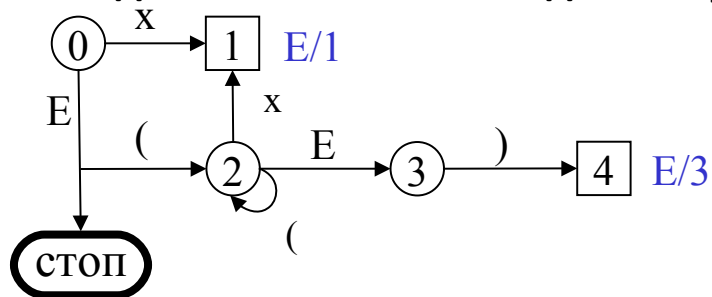
Оставащ вход

E ))



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

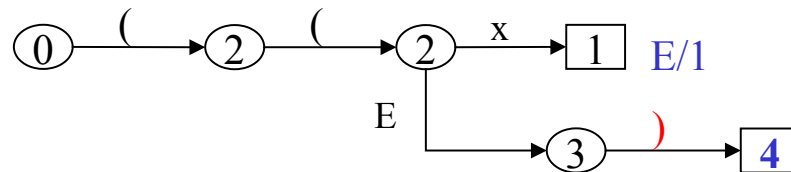
Преминатите състояния се съхраняват в стек

Стек

4
3
2
2
0

Оставащ вход

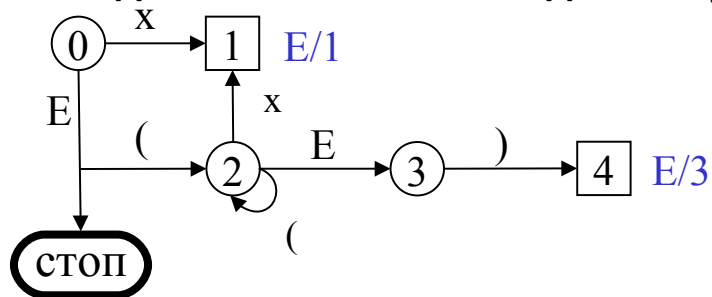
**)**)





# Действие на стеков автомат

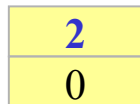
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

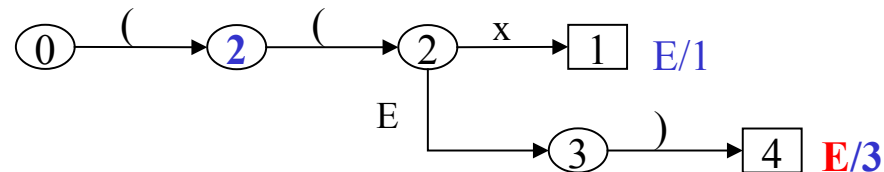
Преминатите състояния се съхраняват в стек

Стек



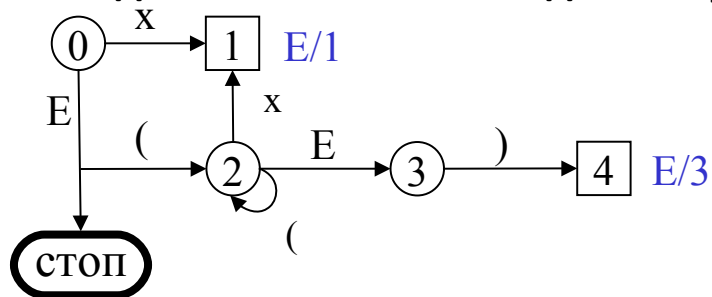
Оставащ вход

E )



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

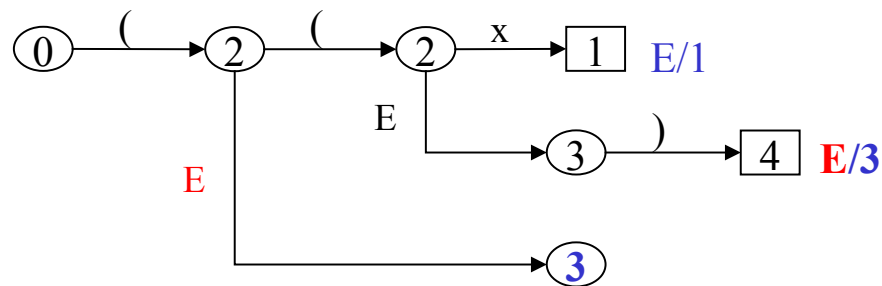
Преминатите състояния се съхраняват в стек

Стек

3
2
0

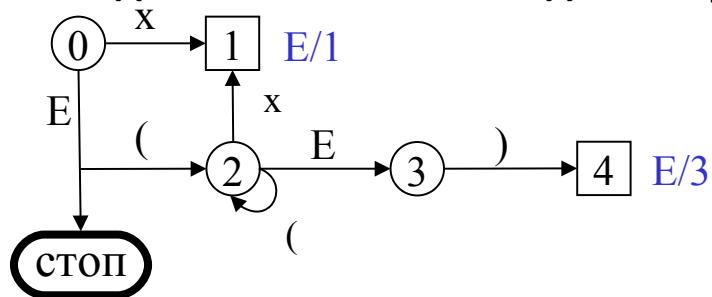
Оставащ вход

E )



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

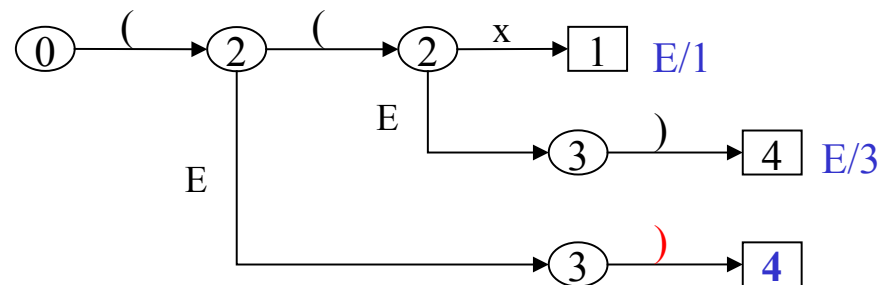
Преминатите състояния се съхраняват в стек

Стек

4
3
2
0

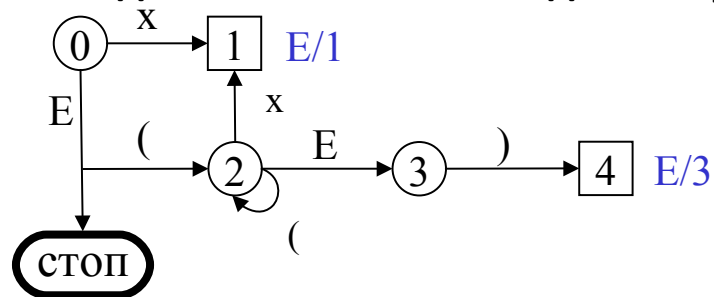
Оставащ вход

)



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: **((x))**

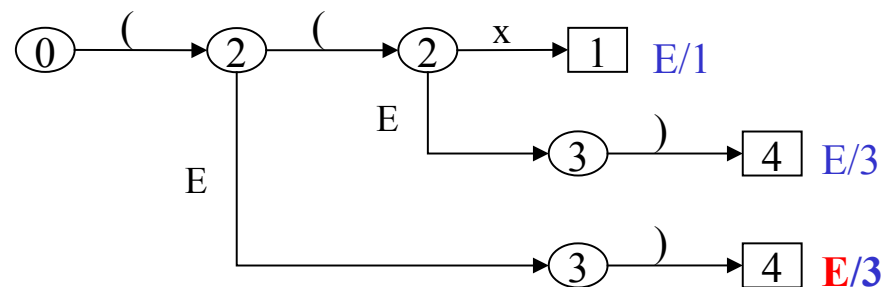
Преминатите състояния се съхраняват в стек

Стек

0

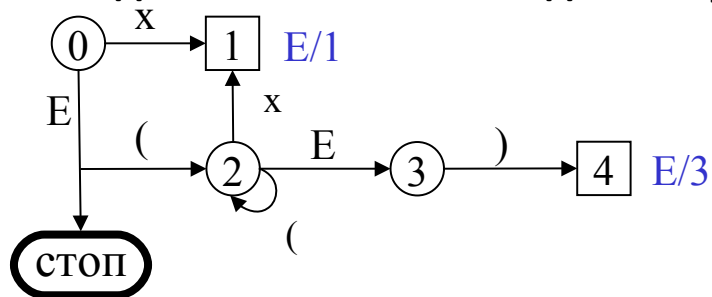
Оставащ вход

E



# Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

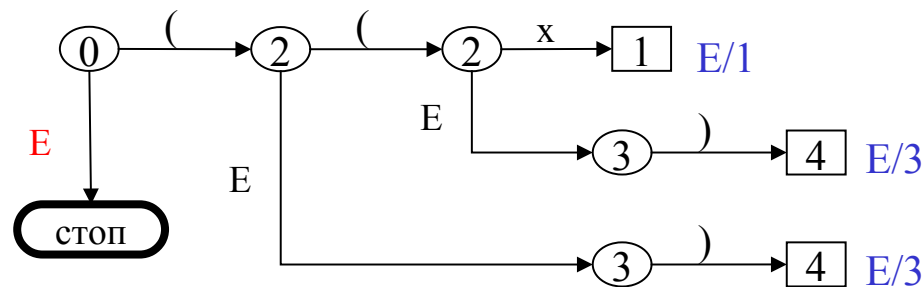
Преминатите състояния се съхраняват в стек

Стек



Оставащ вход

E



# Ограничения на КСТГ

**Контекстно-свободните граматики не могат да изразяват условия, зависещи от контекста. Например:**

- ❖ Всяка променлива трябва да бъде декларирана преди да бъде използвана  
*Декларацията принадлежи на контекста на използване. Операторът  $x = 3$ ; може да бъде правилен или грешен в зависимост от неговия контекст*
- ❖ Операндите на израз трябва да бъдат от един и същи тип  
*Типовете се определят в декларациите, което принадлежи на контекста на използване*

## Възможни решения

- ❖ Използване на контекстно-чувствителни граматики  
*Прекалено сложно*
- ❖ Проверка на контекстните условия по време на семантичния анализ



# Условия, зависещи от контекста

Семантични ограничения, които са определени за всяко произведение.

Например в SimpleC# такива са:

Statement = Location '=' Expr ';'.

- ❖ *Location* трябва да бъде променлива или масив
- ❖ Типът на израза “Expr” трябва да бъде съвместим с типа на *Location*

Location<sub>1</sub> = Location<sub>2</sub> '[' Expr ']'.

- ❖ Типът на *Location<sub>2</sub>* трябва да бъде масив
- ❖ Типът на израза “Expr” трябва да бъде *int*

# Свойство $LL(1)$

## Предусловие за метода на рекурсивното спускане

$LL(1)$  означава, че граматиката може да бъде анализирана от ляво надясно с ляв каноничен извод (най-левият НТС се опростява първи) с преглед 1 лексема напред (Left to right with Left-canonical derivations and 1 lookahead symbol)

## Определение:

1. Една граматика е  $LL(1)$ , ако всичките ѝ произведения са  $LL(1)$
2. Едно произведение е  $LL(1)$ , ако за всичките алтернативи

$\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  е изпълнено:

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset \text{ (за всяко } i \text{ и } j)$$

## С други думи

- ❖ Терминалните стартови символи на всички алтернативи на произведение по двойки трябва да бъдат празни множества
- ❖ Парсерът винаги трябва да може да избере една от алтернативите като гледа следващата лексема





# Премахване на LL(1) конфликти

## Факторизация:

```
IfStatement = 'if' '(' Expr ')' Statement  
             | 'if' '(' Expr ')' Statement 'else' Statement.
```

```
IfStatement = 'if' '(' Expr ')' Statement ( | 'else' Statement ).
```

```
IfStatement = 'if' '(' Expr ')' Statement ['else' Statement ].
```

За да се избере  
алтернатива, трябва да се  
прочете повече от 1  
лексема напред

Изнасят се общите  
части

В EBNF

## Понякога НТС трябва да бъдат вградени преди факторизация

```
Statement = Location '=' Expr ';'   
           | ident '(' [ ActualParameters ] ')' ';' .  
Location = ident { '.' ident } .
```

```
Statement = ident { '. ident } '=' Expr ';'   
           | ident '(' [ ActualParameters ] ')' ';' .
```

```
Statement = ident ( { '.' ident } '=' Expr ';' | '(' [ ActualParameters ] ')' ';' ) .
```

Вгражда се Location в  
Statement

В EBNF



# Премахване на лява рекурсия

**Лявата рекурсия е винаги конфликт при LL(1):**

Например:

```
IdentList = ident | IdentList ',' ident.
```

Генерира:

ident

ident ',' ident

ident ',' ident ',' ident

...

Може винаги да се замени с итерация

```
IdentList = ident { ',' ident }.
```

Така ',' е стоп  
символ за цикъла



# Скрити $LL(1)$ конфликти

Опциите и итерациите в EBNF са скрити алтернативи

$A = [ \alpha ] \beta.$      $\Leftrightarrow$      $A = \alpha \beta \mid \beta.$      $\alpha$  и  $\beta$  са произволни изрази от EBNF

Правила:

$A = [ \alpha ] \beta.$      $\text{First}(\alpha) \cap \text{First}(\beta)$  трябва да бъде  $\emptyset$

$A = \{ \alpha \} \beta.$      $\text{First}(\alpha) \cap \text{First}(\beta)$  трябва да бъде  $\emptyset$

$A = \alpha [ \beta ].$      $\text{First}(\beta) \cap \text{Follow}(A)$  трябва да бъде  $\emptyset$

$A = \alpha \{ \beta \}.$      $\text{First}(\beta) \cap \text{Follow}(A)$  трябва да бъде  $\emptyset$

$A = \alpha \mid .$      $\text{First}(\alpha) \cap \text{Follow}(A)$  трябва да бъде  $\emptyset$



# Елементи на множеството *First*

## Алгоритъм за намиране:

1. За всеки НТС  $A \in G$  (контекстно-свободна граматика)  $First(A) = \emptyset$
2. За всяко правило  $A = \alpha$  ( $\alpha$  е съвкупност от ТС и НТС) се добавят всичките елементи на  $First(\alpha)$  към  $First(A)$ . Това са:
  - 2.1. ако  $\alpha = \epsilon$ , добавя се  $\epsilon$  към  $First(A)$
  - 2.2. ако първият символ в  $\alpha$  е ТС (напр.  $a$ ), добавете го ( $a$ ) към  $First(A)$
  - 2.3. ако  $\alpha = A_1 \alpha'$  за някой НТС  $A_1$  и  $First(A)$  не съдържа  $\epsilon$ , тогава се добавят всички елементи на  $First(A_1)$  към елементите на  $First(A)$
  - 2.4. ако  $\alpha = A_1 \alpha'$  за някой НТС  $A_1$  и  $First(A)$  съдържа  $\epsilon$ , тогава се добавят всички елементи на  $First(A_1)$  с изключение на  $\epsilon$  към елементите на  $First(A)$  и рекурсивно се добавят всички елементи на  $First(\alpha')$  към  $First(A)$
3. Ако са направени промени в стъпка 2, върни се в стъпка 2



# Пример за елементи на First

Дадена е граматиката:

[0]  $S' = S \$$ .

[1]  $S = A B$ .

[2]  $S = C f$ .

[3]  $A = e f$ .

[4]  $A = \varepsilon$ .

[5]  $B = h g$ .

[6]  $C = D D$ .

[7]  $C = f i$ .

[8]  $D = g$

Таблицата с елементите на First

Нетерминали	First ел.
$S'$	e, f, g, h
$S$	e, f, g, h
$A$	e, $\varepsilon$
$B$	h
$C$	f, g
$D$	g



# Елементи на множеството Follow

## Алгоритъм за намиране:

1. Намерете  $\text{First}(A)$ ,  $A \in G$
2. За всеки НТС  $A \in G$  (контекстно-свободна граматика)  $\text{Follow}(A) = \emptyset$
3. За всяко правило  $A = \alpha$  ( $\alpha$  е съвкупност от ТС и НТС) и за всеки НТС  $A_1$  трябва да се направи:
  - 3.1. ако правилото е от вида  $A = \alpha A_1 \beta$  ( $\alpha, \beta$  са низове от ТС и НТС, които е възможно да са празни) и  $\text{First}(\beta)$  не съдържа  $\epsilon$ , добавят се всички елементи от  $\text{First}(\beta)$  към  $\text{Follow}(A_1)$
  - 3.1. ако правилото е от вида  $A = \alpha A_1 \beta$  ( $\alpha, \beta$  са низове от ТС и НТС, които е възможно да са празни) и  $\text{First}(\beta)$  съдържа  $\epsilon$ , добавят се всички елементи от  $\text{First}(\beta)$  без  $\epsilon$  към  $\text{Follow}(A_1)$  и всички елементи на  $\text{Follow}(A)$  се добавят към  $\text{Follow}(A_1)$
4. Ако са направени промени в стъпка 3, върни се към стъпка 3



# Пример за елементи на Follow

Дадена е граматиката:

[0]  $S' = S \$$ .

[1]  $S = A B$ .

[2]  $S = C f$ .

[3]  $A = e f$ .

[4]  $A = \varepsilon$ .

[5]  $B = h g$ .

[6]  $C = D D$ .

[7]  $C = f i$ .

[8]  $D = g$

Таблицата с елементите на Follow

Нетерминали      Follow ел.

$S'$	$\emptyset$
$S$	$\$$
$A$	$h$
$B$	$\$$
$C$	$f$
$D$	$f, g$



# “Висящ” Else

## If операторът в C#

```
Statement = 'if' '(' Expr ')' Statement [ 'else' Statement ]  
           | ...
```

## Това е LL(1) конфликт!

$\text{First}(\text{'else' Statement}) \cap \text{Follow}(\text{Statement}) = \{\text{'else'}\}$

## Това е двусмислие, което не може да бъде премахнато

Statement  
┌───┐  
Statement  
┌──────────┐  
if (expr1) if (expr2) stat1; else stat2;  
└──────────┘  
Statement  
└───┘  
Statement

*Могат да се построят две различни синтактични дървета!*





# Може ли LL(1) конфликтите да се пренебрегнат

## LL(1) конфликтът е само предупреждение

Парсерът избира първата подходяща алтернатива

```
A = a b c
   | a d.
```

Ако следващата лексема е 'a', парсерът избира тази алтернатива

## Пример: “Висящ” Else

```
Statement = 'if' '(' Expr ')' Statement [ 'else' Statement ]
           | ... .
```

Ако следващата лексема е 'else' парсерът анализира тази опция, т.е. else-а принадлежи на най-вътрешния if

```
if (expr1) if (expr2) stat1; else stat2;
```

Statement

Statement

В случая точно това искаме



# Други изисквания за граматика

(необходими при парсване)

## Пълнота

За всеки НТС трябва да има извод:

$A = a B C .$

Грешка

$B = b b .$

Няма извод за C

## Изводимост

Всеки НТС трябва бъде изводим (директно или косвено) в съвкупност от ТС

$A = a B | c .$

Грешка

$B = b B .$

B не може да бъде сведен до съвкупност от ТС

## Ацикличност

НТС не трябва да бъде изводим в себе си (директно или косвено) ( $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \dots \Rightarrow A$ )

$A = a b | B .$

Грешка

$B = b b | A .$

Тази граматика е циклична защото  $A \Rightarrow B \Rightarrow A$



# *Управление на Грешки* *(diagnostics)*



# *Видове*

1. Паник режим
2. Управление чрез общи синхронизиращи лексеми
3. Управление чрез специални синхронизиращи лексеми

# Цели на управлението на грешки

## Изисквания:

- ❖ Парсера трябва да открие възможно най-много грешки по време на компилация;
- ❖ Парсера не трябва да влиза в безкраен цикъл;
- ❖ Добре е управлението на грешки да не забавя парсера;
- ❖ Добре е управлението на грешки да не увеличава кода на парсера;

# 1. Паник режим

Парсера приключва работата си при откриване на грешка

```
public static void Error(Token token, String msg) {  
    throw new Exception(  
        string.Format("Error at line {0}, column {1}: {2}",  
            token.Line, token.Col, msg)  
    );  
}
```

**Предимства:**

- ❖ Лесен за реализация;
- ❖ Ефективен за малки командни езици и за интерпретатори;

**Недостатъци:**

- ❖ Не е подходящ за високо продуктивни компилатори;



# 1. Паник режим – пример

Например: A = a b c .

```
public bool IsA() {  
    if (!Check("a")) Error(token, "'a' expected");  
    if (!Check("b")) Error(token, "'b' expected");  
    if (!Check("c")) Error(token, "'c' expected");  
    return true;  
}
```

## Забележки:

- ❖ Най-често съобщаването на грешки изписва и реда и колоната, на които е срещната грешката. Те могат да бъдат получени от полетата на **token**, където се пази последната разпозната лексема, заедно с нейните характеристики (част от които са и реда и колоната, където се среща лексемата);
- ❖ Текста на грешката трябва да подсказва на програмиста ясно естеството на проблема;



## 2. Управление чрез общи синхронизиращи лексеми

### Пример:

очакван вход:                    a b c d ...

реален вход:                    a **x y** d ...

### Възстановяване (синхронизира оставащия вход с граматиката):

**1. Намират се синхр. лексеми, от които парсерът може да продължи след грешка**

*Кои са възможните лексеми, от които парсерът може да продължи в горната ситуация?  
с следва b (очакващ се при възникване на грешката), d следва b и c, ...*

*Синхронизиращи лексеми са {c, d, ...}*

**2. Прескачат се входните лексеми докато не се намери синхр. лексема**

*Прескачат се x и y, a d е синхр. лексема, от която парсерът може да продължи*

**3. Прескачане на парсера до позицията в граматиката, от където може да продължи**





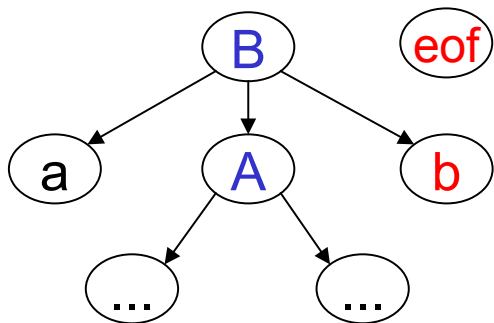
# Пресмятане на синхр. лексеми

Всеки анализиращ метод на нетерминал (A) приема следващите лексеми на нетерминала като параметри:

```
public bool IsA(BitArray sux)
```

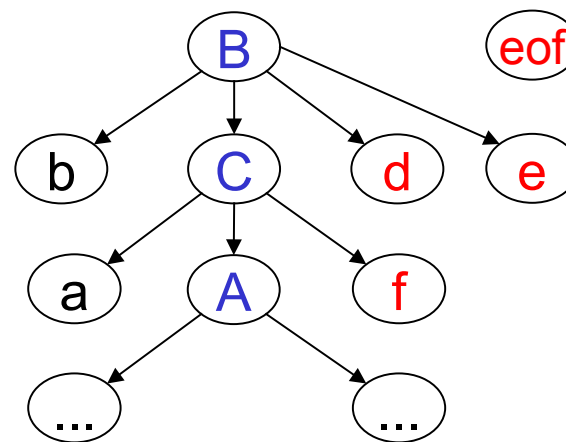
Съдържа наследници на всички НТС, които се обработват в момента

В зависимост от контекста  $sux_A$  може да отразява различни множества



$sux_A = \{b, eof\}$

*sux винаги съдържа eof (наследникът на стартовия символ)*



$sux_A = \{f, d, e, eof\}$

# Управление на терминални символи

Граматика

$A = \dots a s_1 s_2 \dots s_n$

$s_i = TC \cup HTC$

Анализиращо действие

$\text{Check}(a, \text{sux}_A \cup \text{First}(s_1) \cup \text{First}(s_2) \cup \dots \cup \text{First}(s_n))$

*може да бъде изчислено по време на компилация  
трябва да бъде изчислено по време на изпълнение*

Например:  $A = a b c .$

```
public bool IsA(BitArray sux) {  
    if (!Check("a", Add(sux, fs1))) Error(...);  
    if (!Check("b", Add(sux, fs2))) Error(...);  
    if (!Check("c", sux)) Error(...);  
    return true;  
}
```

# Управление на терминални символи

```
static BitArray fs1 = new BitArray();  
fs1[b] = true;  
fs1[c] = true;
```

Например: A = a b c .

```
public bool IsA(BitArray sux) {  
    if (!Check("a", Add(sux, fs1))) Error(...);  
    if (!Check("b", Add(sux, fs2))) Error(...);  
    if (!Check("c", sux)) Error(...);  
    return true;  
}
```



# Управление на терминални символи

```
static BitArray Add (BitArray a, BitArray b) {  
    BitArray c = (BitArray) a.Clone();  
    c[b] = true;  
    return c;  
}
```

Например: A = a b c .

```
public bool IsA (BitArray a, BitArray b, BitArray c, BitArray sux) {  
    if (!Check ("a", Add (a, sux, fs1)) Error (...));  
    if (!Check ("b", Add (a, sux, fs2)) Error (...));  
    if (!Check ("c", sux) Error (...));  
    return true;  
}
```

# Управление на нетерминални символи

Граматика

$A = \dots B s_1 s_2 \dots s_n$

Анализиращо действие

$B(a, \text{sux}_A \cup \text{First}(s_1) \cup \text{First}(s_2) \cup \dots \cup \text{First}(s_n))$

$s_i = TC \cup HTC$

Например:  $A = a B c .$   
 $B = b b .$

```
public bool IsA(BitArray sux)
{
    Check("a", Add(sux, fs3));
    IsB(Add(sux, fs4));
    Check("c", sux);
}
```

$fs4 = \{c\}$

$fs3 = \{b, c\}$

```
public bool IsB(BitArray sux)
{
    Check("b", Add(sux, fs5));
    Check("b", sux);
}
```

$fs5 = \{b\}$

Анализиращият метод на стартовия символ  $S$  се казва  $IsS(fs_0)$ , където  $fs_0 = \{eof\}$



# Прескачане на невалидните входни лексеми

Грешките се откриват в Check:

```
public void Check(string expected, BitArray sux) {  
    if (token == expected) ReadNextToken();  
    else Error ("Очаква се" + expected, sux);  
}
```

След отпечатване на съобщение за грешка входните лексеми се прескачат докато не се намери синхронизираща лексема

```
public void Error(string msg, BitArray sux) {  
    Console.WriteLine("Линия {0}, колона {1}:{2},",  
        token.Line, token.Col, msg);  
  
    errors++;  
    while (!sux[token.number]) ReadNextToken(); //докато лекс. ∉  
    sux //лекс. ∈ sux  
}
```

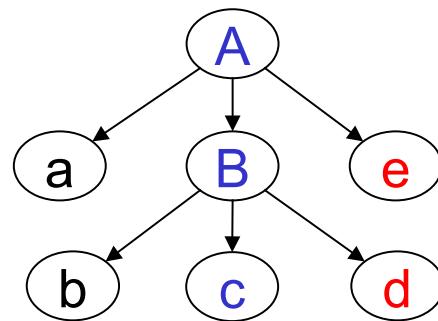
```
public int errors = 0; //броя открити синтактични грешки
```



# Синхронизиране с граматиката

Пример:  $A = a B e .$   
 $B = b c d .$

```
public void A(BitArray sux) {  
    Check("a", Add(sux, fs1));  
    B(Add(sux, fs2));  
    Check("e", sux);  
}  
  
public void B(BitArray sux) {  
    Check("b", Add(sux, fs3));  
    Check("c", Add(sux, fs4));  
    Check("d", sux);  
}
```



Вход:  $a \quad b \quad x \quad e \quad eof$

Грешката е открита тук синхр. лекс.  $\{d, e, eof\}$

1.  $x$  се прескача;  $token == e$  ( $\in$  синхр. лекс.)
2. Парсера продължава:  $Check("d", sux)$ ;
3. Открива грешка отново; синхр. лекс =  $\{e, eof\}$
4. Нищо не се прескача;  $token == e$  ( $\in$  синхр. лекс.)
5. Парсера излиза от  $B()$  и прави  $Check("e", sux)$ ;
6. Успешно възстановяване.

След грешката парсерът “прескача” напред докато не стигне до точка в граматиката, където намерената синхронизираща лексема е валидна

# Прескачане на фалшиви грешки

По време на възстановяването на грешки, парсерът извежда фалшиви грешки

Проблемът се решава чрез проста евристика

Ако са разпознати по-малко от 3 лексеми след регистриране на последната грешка, се предполага, че новооткритата грешка е фалшива. Фалшивите грешки не се извеждат.

```
public int errDist = 3;

public void ReadNextToken() { //...; errDist++;}

public void Error(string msg, BitArray sux) {
    if (errDist >= 3) {
        Console.WriteLine("Линия {0}, колона {1}:{2},",
            token.Line, token.Col, msg);
        errors++;
    }
    while (!sux[token.number]) ReadNextToken();
    errDist = 0;
}
```



# Управление на алтернативи

$A = \alpha \mid \beta \mid \gamma$   $\alpha, \beta, \gamma$  са произволни изрази в EBNF

```
public void A(BitArray sux) {  
    // проверката за грешки е вече направена така че парсерът може да се  
    // синхронизира с началните символи на алтернативите в случай на грешка  
  
    if (token  $\notin$  First( $\alpha$ )  $\cup$  First( $\beta$ )  $\cup$  First( $\gamma$ ))  
        Error("Невалиден A", sux  $\cup$  First( $\alpha$ )  $\cup$  First( $\beta$ )  $\cup$  First( $\gamma$ ));  
    // token е една от алтернативите или е допустим наследник на A  
    else {анализирай  $\gamma$ } // няма проверка за грешки тук, всички грешки са  
        // били вече изведени  
}
```

$First(\alpha) \cup First(\beta) \cup First(\gamma)$  може да бъде предварително изчислено по време на компилация  
 $sux \cup First(\alpha) \cup First(\beta) \cup First(\gamma)$  трябва да бъде изчислено по време на изпълнение



# Управление на опции

Опции:

$A = [\alpha] \beta$ .

```
public void A(BitArray sux) {
    /*проверката за грешки е вече направена така че
    парсерът може да се синхронизира с  $\alpha$  в случай на
    грешка*/
    if (token  $\notin$  First( $\alpha$ )  $\cup$  First( $\beta$ ))
        Error("...", sux  $\cup$  First( $\alpha$ )  $\cup$  First( $\beta$ ));
    //token е  $\alpha$  или  $\beta$  или е допустим наследник на A
    if (token  $\in$  First( $\alpha$ )) {анализирай  $\alpha$ }
    {анализирай  $\beta$ }
}
```

# Управление на итерации

Итерации:

$A = \{ \alpha \} \beta .$

```
public void A(BitArray sux) {  
    while (true) {  
        //влиза се в цикъла дори и token  $\notin$  First( $\alpha$ )  
        if (token  $\in$  First( $\alpha$ )) {анализирай  $\alpha$ }  
        else if (token  $\in$  First( $\beta$ )  $\cup$  sux) break;  
        else Error("...", sux  $\cup$  First( $\alpha$ )  $\cup$  First( $\beta$ ));  
    }  
    {анализирай  $\beta$ }  
}
```

# Пример

A = a B | b {c d}.

B = [b] d.

```
public void A(BitArray sux) {
    if (token != "a" || token != "b")
        Error("...", Add(sux, fs1)); //fs1 = {a, b}
    if (token == "a") {
        ReadNextToken();
        B(sux);
    } else if (token == "b") {
        ReadNextToken();
        while (true) {
            if (token == "c") {
                ReadNextToken();
                Check ("d", Add(sux, fs2)); //fs2 = {c}
            } else if (sux[token.number]) break;
            else Error("...", Add(sux, fs2));
        }
    }
}
```

```
public void B(BitArray sux) {
    if (token != "b" && token != "d")
        Error("...", Add(sux, fs3)); //fs3 = {b, d}
    if (token == "b") ReadNextToken();
    Check (d, sux);
}
```



# Оценка на управление със общи синхр. лексеми

## Предимства:

- ❖ Системно приложима

## Недостатъци:

- ❖ Забавя анализа на програми без грешки
- ❖ Увеличава кода на парсера
- ❖ Усложнява реализацията

### 3. Управление чрез специални синхр. лексеми

**Управлението на грешките се прави само на “безопасни” места:**

*Например на места, които започват с ключови думи, които не се срещат на други места в граматиката*

**Например:**

- ❖ Начало на изречение: if, while, do, ...
  - ❖ Начало на декларация: public, static, void, ...
- } множ. синхр. лекс

Възниква проблем: Може ли Ident да бъде в множествата?

Не, защото Ident може да има и на двете позиции!

(Ident не е “безопасна” лексема)

### 3. Управление чрез специални синхр. лексеми

**Кодът, който трябва да се добави към точките за синхронизация**

```
...множество на спец. лекс. в тази точка на синхронизация
if (token ∉ expectedSymbols) {
    Error("...");
    //няма наследени множества, които да се премятат непрекъснато.
    while (token ∉ (expectedSymbols ∪ {eof})) ReadNextToken();
} // + eof за да избегнем безкраен цикъл
...
```

- ❖ Няма наследени множества, които да се предават на методите за анализ;
- ❖ Множествата могат да бъдат изчислени по време на компилация;
- ❖ След откриване на грешка парсера “прескача” до следващата точка на синхронизация;



# Пример

Statement = 'if' '(' Expr ')' Statement ['else' Statement] ';' | 'while' ...  
Синхронизация в началото на изречение (Statement)

```
public void Statement() {
    if (!firstStat[token.number]) {
        Error("...");
        while (!firstStat[token.number] && token != EOF)
            ReadNextToken();
    }
    if (token == "if") {
        ReadNextToken();
        Check("("); Expr(); Check(")"); Statement();
        if (token == "else") {
            ReadNextToken();
            Statement();
        } else if (token == "while") ...
    }
}
```

*firstStat съдържа  
множеството на  
специалните  
синхронизиращи лексеми*

*Останалата част на  
парсера не се променя*

```
public void Error(string msg) {
    if (errDist >= 3) {
        Console.WriteLine("Линия {0}, колона {1}:{2},", token.Line, token.Col, msg);
        errors++;
    }
    errDist = 0; // евристиката може да се приложи и тук...
}
```



# Пример за възстановяване

```
public void Statement() {
    if (!firstStat[token.number]) {
        Error("...");
        while (!firstStat[token.number] && token != EOF)
            ReadNextToken();
    }
    if (token == "if") {
        ReadNextToken();
        Check("("); Expr(); Check(")"); Statement();
        if (token == "else") {
            ReadNextToken();
            Statement();
        } else if (token == "while") ...
    }
}
```

```
public void Error(string msg) {
    if (errDist >= 3) {
        Console.WriteLine("...");
        errors++;
    }
    errDist = 0;
}
```

```
public void Check(string
expected) {
    if (token == expected)
        ReadNextToken();
    else Error ("Очаква се" +
               expected);
}
```

Неправилен вход: if a>b then max = a;

<i>token</i>	<i>action</i>
IF	ReadNextToken(); IF $\in$ <i>firstStatement</i> $\Rightarrow$ ok
a	Check("("); грешка: очаква се '(' Expr(); разпознава a > b
THEN	Check(")"); грешка: очаква се ')' Statement(); THEN не се разпознава $\Rightarrow$ грешка без съобщение THEN се прескача; синхронизация с <i>ident</i> (ако е в <i>firstStat</i> )

max



# Синхронизация в началото на итерация

Например: Block = '{' {Statement} '}'.

Шаблона в този случай е:

```
public void Block () {  
    Check ("{" );  
    while (firstStat[token.number])  
        Statement ();  
    Check ("}" );  
}
```

*Проблем: Ако следващата лексема не разпознае Statement цикълът не се изпълнява и следователно синхронизационната точка не се достига.*

# Синхронизация в началото на итерация

Например: Block = '{ {Statement} }'.

По-добре е да се синхронизира в края на итерацията

```
public void Block() {
    Check("{");
    while (true) {
        if (token ∈ First(Statement)) Statement();
        else if (token ∈ {rbrace, eof}) break;
        else {
            Error("invalid start of Statement");
            do
                ReadNextToken ();
            while (token ∈ (First(Statement) ∪ {rbrace, eof}));
        }
    }
    Check("}");
}
```

Не зависи от синхронизация в **Statement()**



# Оценка на управление със спец. синхр. лексеми

## Предимства:

- ❖ Не забавя анализа на програми без грешки
- ❖ Не увеличава кода на парсера
- ❖ Проста реализация

## Недостатъци:

- ❖ Изисква опит и фина настройка

*Въпроси?*  
*apenev@uni-plovdiv.bg*

