Методи на Транслация

Лексикален анализ.

Синтактичен анализ.

Управление на грешки.

доц. д-р Александър Пенев

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)

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 и др.;

Регулярни граматики

Определение:

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

A = a . a, b ∈ TC

A = b B . A, B ∈ 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 ⇒ Letter Other ⇒ Letter Letter Other ⇒ Letter Letter Letter Other ⇒ Letter Letter Letter Digit

Регулярни граматики

Друго определение:

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

Пример: *Граматика за идентификаторите*

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

Детерминиран краен автомат (ДКА)

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

Определение:

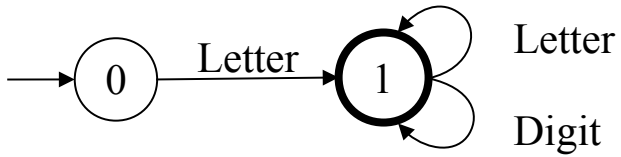
Детерминиран краен автомат е наредена петорка (S, I, δ, s0, F), където:

- S – вътрешна азбука (множество състояния);
- I – външна азбука (множество от входни символи);
- δ: S x I → S – функция на преход;
- s0 – 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:



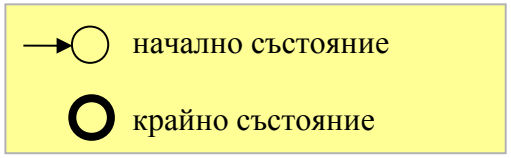


Таблица на състоянията

δ	Letter	Digit
s0	s1	грешка
s1	s1	s1

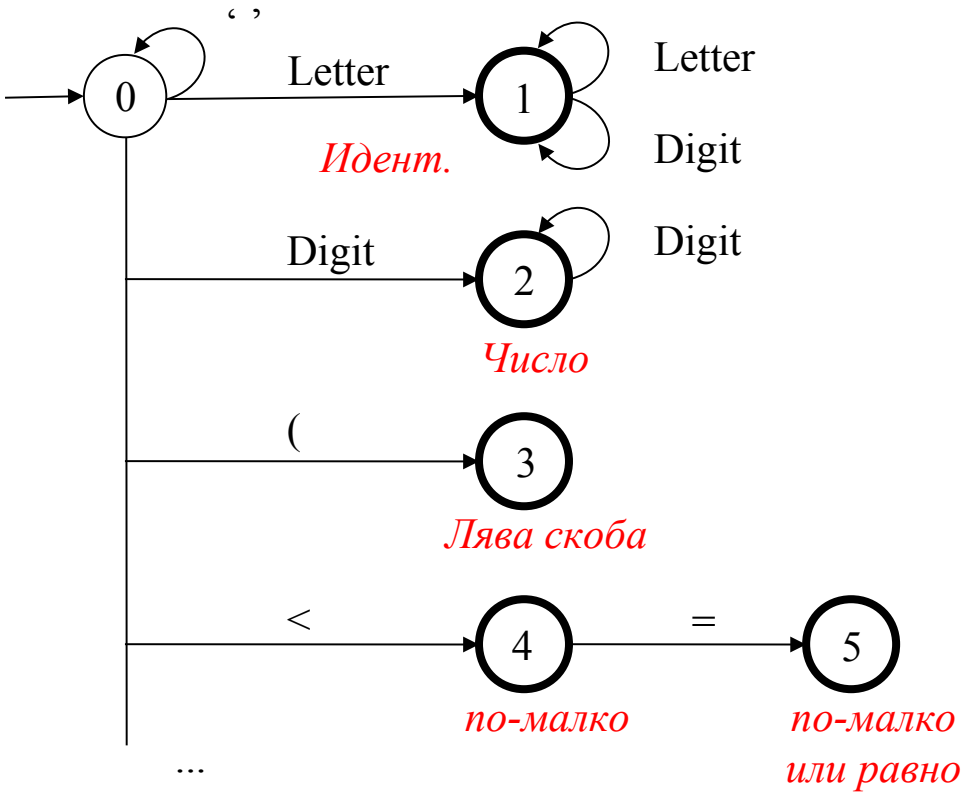
"краен", защото δ може да бъде записана явно

ДКА разпознава изречение:

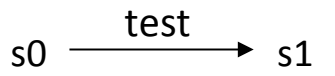
- *ако е в някое крайно състояние;*
- *и ако входните символи са напълни консумирани или няма възможен преход със следващия символ;*

Скенерът като ДКА

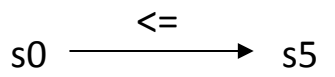
Скенерът може да бъде разглеждан като голям ДКА.



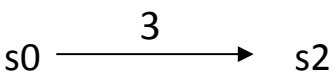
Примерен вход: test <= 3



- Няма преход при ‘ ’ към s1
- Лексемата “Идент.” е разпозната



- Прескача интервалите в началото
- Не спира в s4
- Няма преход при ‘ ’ в s5
- Лексемата “по-малко” е разпознава



- Прескача интервалите в началото
- Няма преход при ‘ ’ в s2
- Лексемата “Число” е разпознава

След всяка разпозната лексема скенера започва от s0 отново!

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 на подпрограма;
- Съставни оператори;

Контекстно-свободни граматики

Проблем

Регулярните граматики не могат да се справят с централната рекурсия

E = x | ‘(’ E ‘)’.

За подобни случаи са необходими контектстно-свободни граматики.

Контекстно-свободни граматики

Определение

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

A = α. A $\in$ НТС, α 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

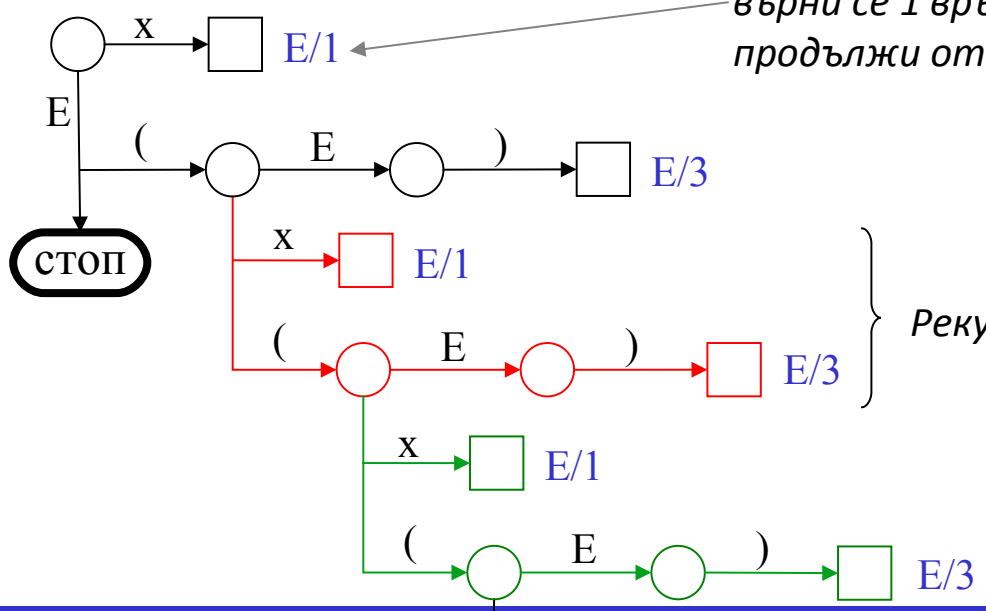
*Контектстно-свободните граматики се разпознават от **стекови автомати***

Стекови автомати

Характеристики:

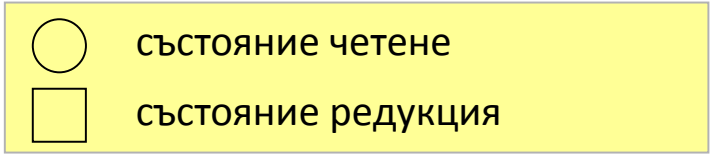
- Позволява преходи с терминални и нетерминални символи;
- Използва стек за да запомни състоянията, през които е преминал;

Например: E = x | ‘(’ E ‘)’.



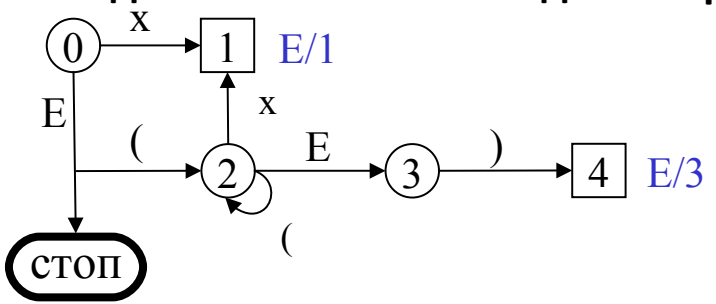
“E” разпознат
върни се 1 връх и
продължи оттам с “E”

Рекурсивно извикване на “E”



Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

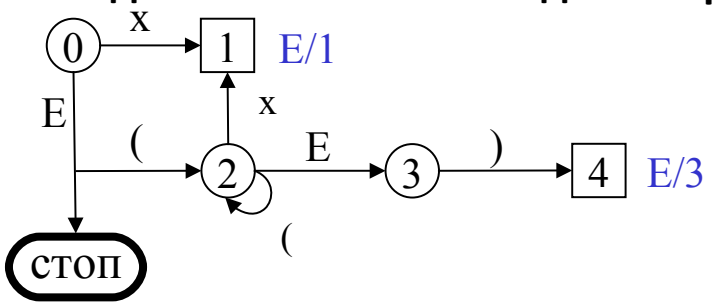
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
0	((x))

0

Действие на стеков автомат

Разгледания автомат може да се опрости:



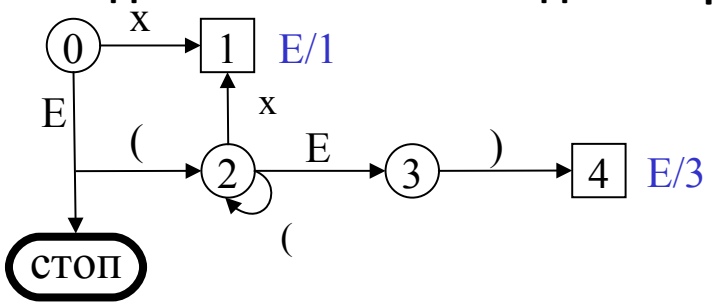
Нека разгледаме входа: ((x))

Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
2 0	((x))

Действие на стеков автомат

Разгледания автомат може да се опрости:



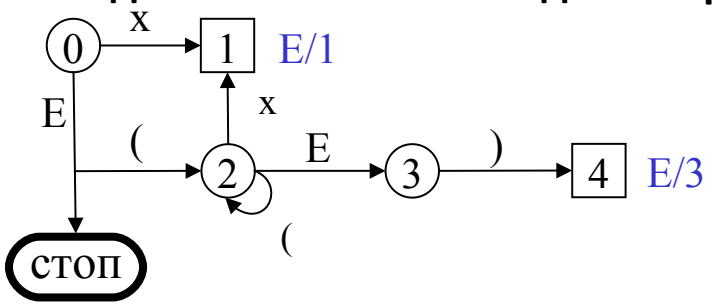
Нека разгледаме входа: ((x))

Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
2 0	((x))

Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

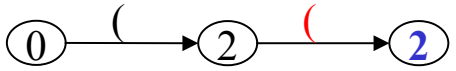
Преминатите състояния се съхраняват в стек

Стек

2
2
0

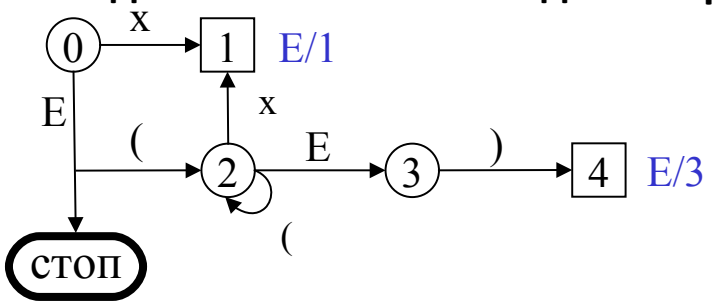
Оставащ вход

(x))



Действие на стеков автомат

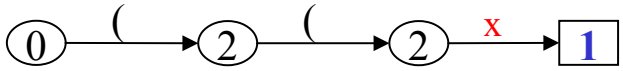
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

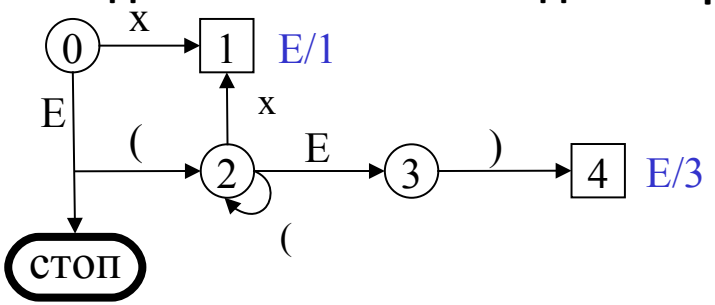
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
1	x))
2	
2	
0	



Действие на стеков автомат

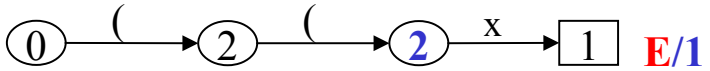
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

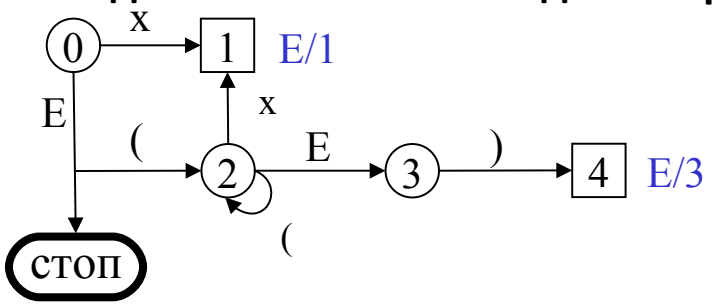
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
2	E))
2	
0	



Действие на стеков автомат

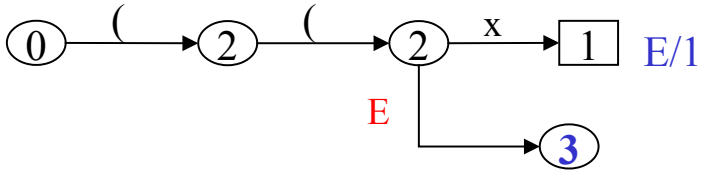
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

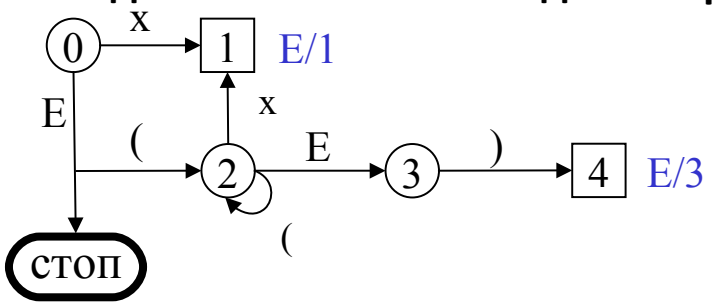
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
3	E))
2	
2	
0	



Действие на стеков автомат

Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

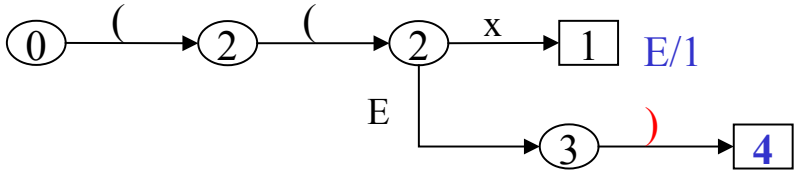
Преминатите състояния се съхраняват в стек

Стек

4
3
2
2
0

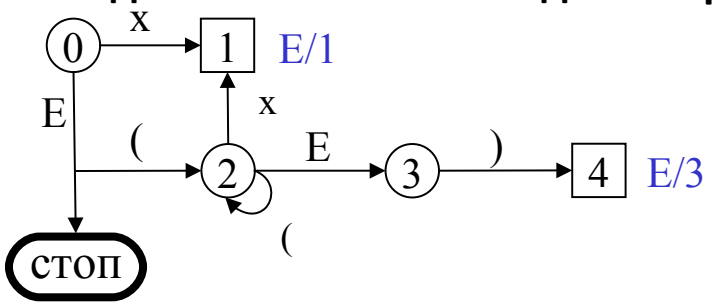
Оставащ вход

))



Действие на стеков автомат

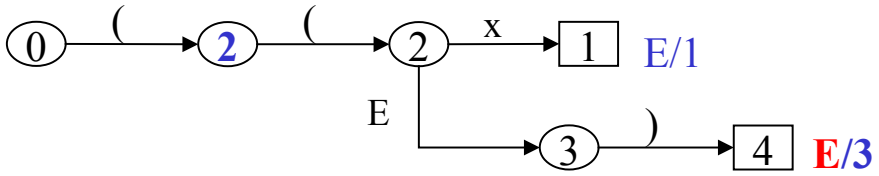
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

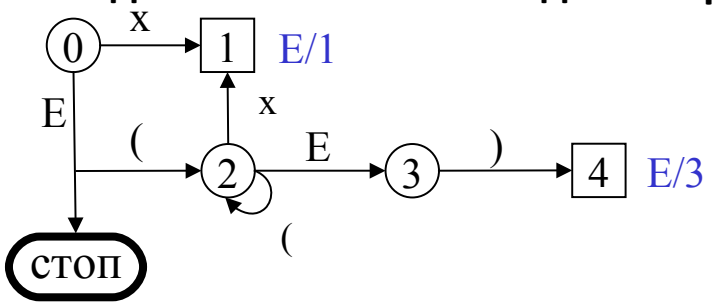
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
2 0	E)



Действие на стеков автомат

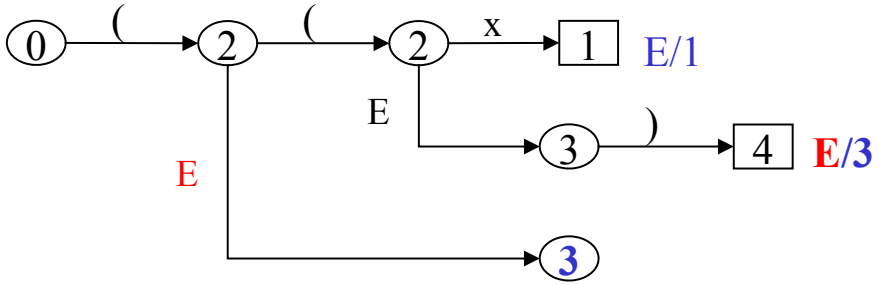
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

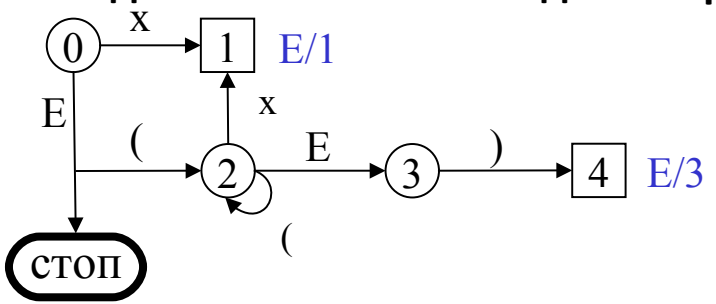
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
3 2 0	E)



Действие на стеков автомат

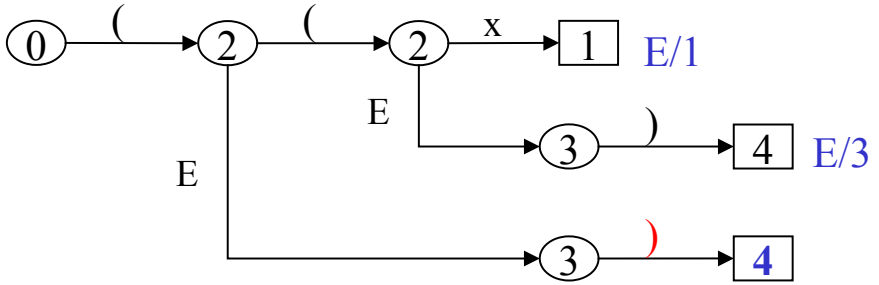
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

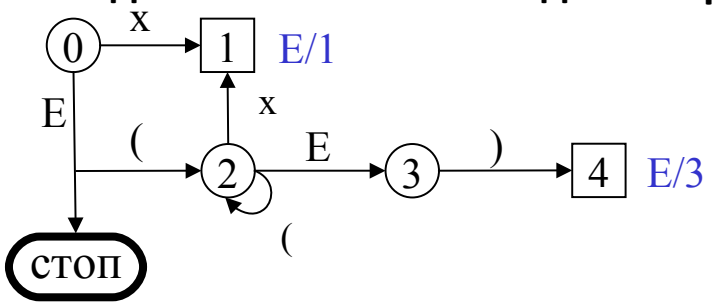
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
4	)
3	
2	
0	



Действие на стеков автомат

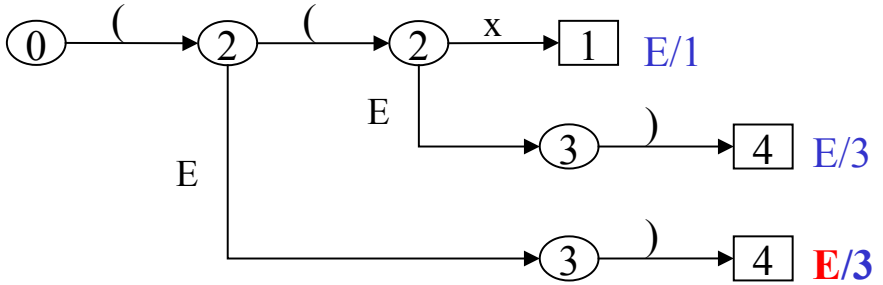
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

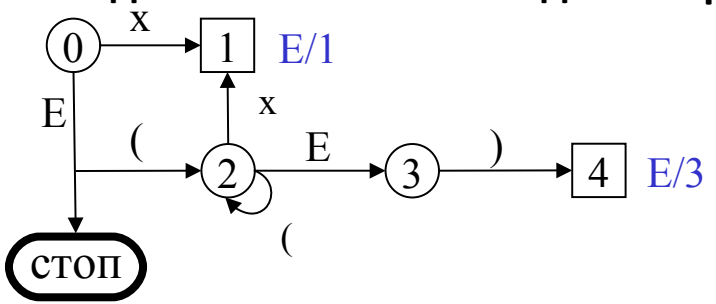
Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
0	E



Действие на стеков автомат

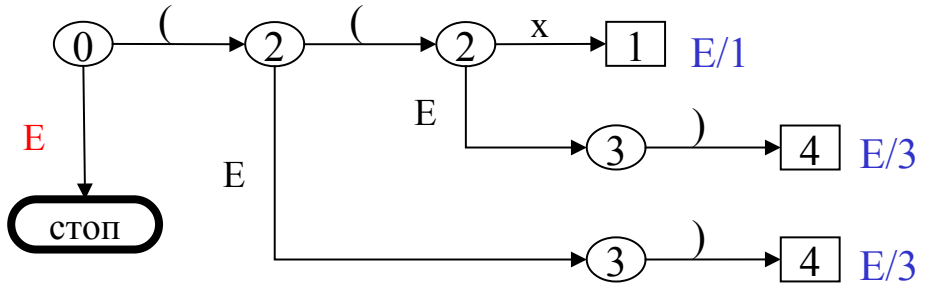
Разгледания автомат може да се опрости:



Нека разгледаме входа: ((x))

Преминатите състояния се съхраняват в стек

Стек	*Оставащ вход*
0	E



Ограничения на КСГ

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 сематичния 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*

$Location_1$ = $Location_2$ '[' Expr ']'.

- Типът на $Location_2$ *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 й произведения са LL(1)
2. Едно произведение е LL(1), ако за всичките алтернативи

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

$First(\alpha_i) \cap First(\alpha_j) = \{\}$ (за всяко i 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.
```

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

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

Изнасят се общите части

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

В EBNF

Понякога НТС трябва да бъдат вградени преди факторизация

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

Вгражда се Location в Statement

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

В EBNF

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

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 = [α] β. ⇔ A = α β | β. α и β са произволни изрази от EBNF

Правила:

A = [α] β.	First(α) ∩ First(β) трябва да бъде {}
A = { α } β.	First(α) ∩ First(β) трябва да бъде {}

A = α [β].	First(β) ∩ Follow(A) трябва да бъде {}
A = α { β }.	First(β) ∩ Follow(A) трябва да бъде {}

A = α | .	First(α) ∩ Follow(A) трябва да бъде {}

Елементи на множеството First

Алгоритъм за намиране:

1. За всеки НТС A∈G (контекстно-свободна граматика) First(A)=∅

2. За всяко правило A = α (α е съвкупност от ТС и НТС) се добавят всичките елементи на First(α) към First(A). Това са:

   2.1. ако $\alpha = \varepsilon$, добавя се ε към First(A)

   2.2. ако първият символ в α 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) <u>не</u> съдържа ε, 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 ε, тогава се добавят всички елементи на First(A_1) с изключение на ε към елементите на First(A) и рекурсивно се добавят всички елементи на First(α') 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 = ε.
[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, ε
B	h
C	f, g
D	g

Елементи на множеството Follow

Алгоритъм за намиране:

1. Намерете First(A), A∈G

2. За всеки НТС A∈G (контекстно-свободна граматика) Follow(A)=∅

3. За всяко правило A = α (α е съвкупност от ТС и НТС) и за всеки НТС A_1 трябва да се направи:

   3.1. ако правилото е от вида A = α $A_1\beta$ (α, β са низове от ТС и НТС, които е възможно да са празни) и First(β) <u>не</u> съдържа ε, добавят се всички елементи от First(β) към Follow(A_1)

   3.1. ако правилото е от вида A = α $A_1\beta$ (α, β са низове от ТС и НТС, които е възможно да са празни) и First(β) съдържа ε, добавят се всички елементи от First(β) без ε към Follow(A_1) и всички елементи на Follow(A) се добавят към 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 = ε.
[5] B = h g.
[6] C = D D.
[7] C = f i.
[8] D = g

Таблицата с елементите на Follow

Нетерминали	Follow ел.
S'	∅
S	$
A	h
B	$
C	f
D	f, g

“Висящ” Else

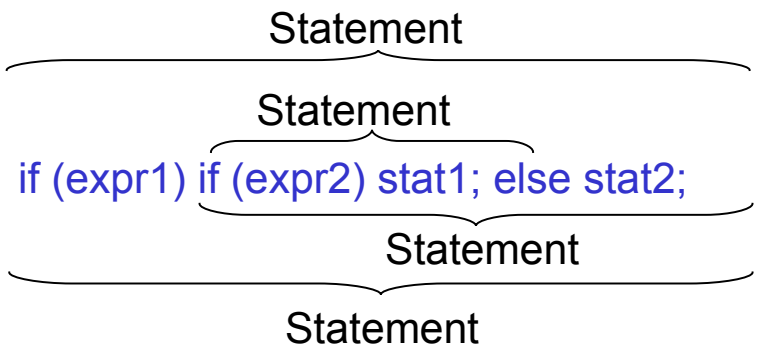
If операторът в C#

```
Statement = ‘if’ ‘(’ Expr ‘)’ Statement [ ‘else’ Statement ]
          | … .
```

Това е LL(1) конфликт!

First(‘else’ Statement) ∩ Follow(Statement) = {‘else’}

Това е двусмислие, което не може да бъде премахнато



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

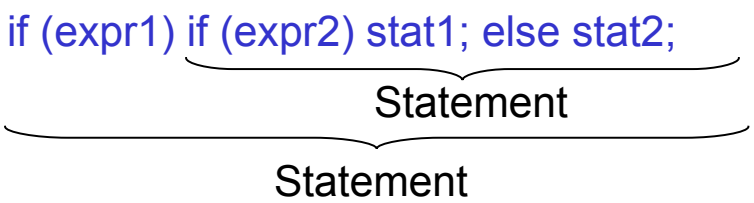
```
A = a b c
  | a d.
```

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



В случая точно това искаме

Други изисквания за граматика

(необходими при парсване)

Пълнота

За всеки НТС трябва да има извод:

```
A = a B C .
B = b b .
```

Грешка

Няма извод за C

Изводимост

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

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

Грешка

B не може да бъде сведен до съвкупност от ТС

Ацикличност

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

```
A = a b | B .
B = b b | A .
```

Грешка

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(int line, int column, String msg) {
  throw new Exception(
    string.Format("Error at line {0}, column {1}: {2}",
                  line, column, msg)
  );
}
```

Предимства:

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

Недостатъци:

- Не е подходящ за високо продуктивни компилатори;

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?

c 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, а d е синхр. лексема, от която парсерът може да продължи

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

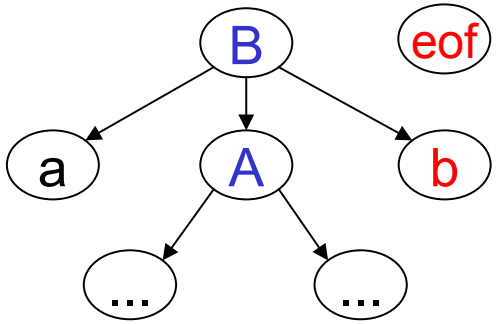
Пресмятане на синхр. лексеми

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

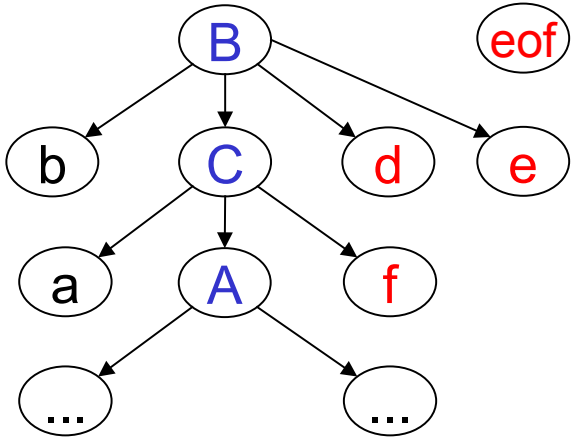
```
public bool IsA(BitArray sux)
```

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_A={f, d, e, eof}

sux винаги съдържа eof (наследникът на стартовия символ)

Управление на терминални символи

Граматика

A = ... a s_1 s_2 ...s_n

s_i=TC ∪ HTC

Анализиращо действие

Check(a, sux_A ∪ First(s_1) ∪ First(s_2) ∪ ... ∪ 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 IsB(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 IsB(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 IsB(BitA      sux) {
  if (!Check("a", A   (sux, fs1))) Error(...);
  if (!Check("b", Add(sux, fs2))) Error(...);
  if (!Check("c", sux)) Error(...);
  return true;
}
```

Управление на нетерминални символи

Граматика

A = ... B s_1 s_2 ...s_n

Анализиращо действие

B(a, sux_A ∪ First(s_1) ∪ First(s_2) ∪ ... ∪ First(s_n))

s_i=TC ∪ 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);
}
```

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

fs4 = {c}

fs3 = {b, c}

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 синхр. 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
```

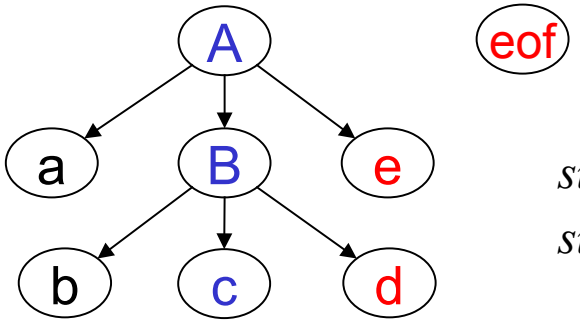
Синхронизиране с граматиката

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);
}
```



$sux_A = \{eof\}$

$sux_B = \{e, eof\}$

Вход: *a* *b* *x* *e* *eof*

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

1. x се прескача; token == e (∈ синхр. лекс.)
2. Парсера продължава: Check(“d”, sux);
3. Открива грешка отново; синхр. лекс = {e, eof}
4. Нищо не се прескача; token == e (∈ 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 синхр. 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 = α | β | γ α, β, γ 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 ∉ First(α) ∪ First(β)∪ First(γ))
     Error("Невалиден A", sux ∪ First(α) ∪ First(β)∪ First(γ));
   // token е една от алтернативите или е допустим наследник на A
   else {анализирай γ} // няма проверка за грешки тук, всички грешки са
                       // били вече изведени
}
```

First(α) ∪ First(β)∪ First(γ) може да бъде преизчислено по време на компилация
sux ∪ First(α) ∪ First(β)∪ First(γ) трябва да бъде изчислено по време на изпълнение

Управление на опции

Опции:

A = [α] β .

```
public void A(BitArray sux) {
   /*проверката за грешки е вече направена така че
     парсерът може да се синхронизира с α в случай на
     грешка*/
   if (token ∉ First(α) ∪ First(β))
     Error("...", sux ∪ First(α) ∪ First(β));
   //token е α или β или е допустим наследник на A
   if (token ∈ First(α)) {анализирай α}
   {анализирай β}
}
```

Управление на итерации

Итерации:

A = { α } β .

```
public void A(BitArray sux) {
   while (true) {
   //влиза се в цикъла дори и token ∉ First(α)
   if (token ∈ First(α)) {анализирай α}
   else if (token ∈ First(β) ∪ sux) break;
   else Error("...", sux ∪ First(α) ∪ First(β));
   }
   {анализирай β}
}
```

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; няма прескачане на лексеми.
   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;

token	*action*	
IF	ReadNextToken();	IF ∈ *firstStatement* ⇒ ok
a	Check("(");	грешка: очаква се '('
	Expr();	разпознава a > b
THEN	Check(")");	грешка: очаква се ')'
	Statement();	THEN не се разпознава ⇒ грешка без съобщение THEN се прескача; синхронизация с *ident* (ако е в *firstStat*)
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